

SMT servo.c Program

Tom Sargent

Introduction:

The servo.c program resides on the "smtots" Linux-hosted computer. This machine has PC104 interfaces to the Traquair (Hunt) HEPC2104 DSP processor which in turn has the A/D interfaces which read the outputs of the SMT shaft encoders. Every 10ms (nominally) the DSP processor sends a position report to the smtots computer which the servo.c program reads. The smtots machine also has ISA bus parallel I/O ports and other A/D and D/A interfaces which interface to the Krupp rack and other SMT control elements.

The servo program also receives position commands from the tracker program (which also runs on smtots). These describe where the telescope should be pointing as a function of time.

With all this input, the servo program is able to read

- 1- the actual telescope position,
- 2- telescope status, and
- 3- the desired telescope position.

The difference of the actual and desired position is the instantaneous tracking error. Knowing the sign and magnitude of this error, it can compute a correction signal to the telescope drive to make it follow the desired track. The only output to the telescope controlling its position from moment to moment is a velocity signal for each axis, so this system is a "velocity loop" servo. The program constantly adjusts the azimuth and elevation velocities to keep the telescope pointed at a desired object as it moves across the sky. The velocities are expressed to the hardware as a $\pm 10\text{v}$ signal which servo generates by using an ISA bus module which has two 16-bit D/A converters.

While controlling the telescope position, the servo program at the same time monitors its inputs for commands from the operator or other programs, and for signals indicating various failures or unsafe conditions, such as limit switches being closed, excessive electrical currents, etc. It can alert the operator or stop the telescope if a serious problem is detected.

Ultimately all commands that move the telescope, regardless of where they originate, end up being performed by servo.

The HEPC2104 DSP processor.

See a separate document for a detailed description of the DSP-resident program.

The Linux system uses a driver from Traquair/Hunt which is called hep2e. It is able to run the ISA bus interface which connects to the DSP processor card's comport 4. The DSP is a Texas Instruments 320c44 chip. The servo program can talk to the hep2e driver directly by doing reads and writes to the driver, or it can use the Hunt Application Program Interface (API) software to do the reads & writes to the DSP. The API software in turn calls that same hep2e driver. It was found that the Hunt API uses asynchronous I/O and this has the side effect of spawning hundreds of tasks per second. Apparently it

does one for each I/O read or write. So, it is recommended to NOT use the Hunt API. The use of the Hunt API is controlled through conditional compilation. Define USE_HUNT_API in servo.c and recompile it, if you want the program to use the Hunt API software. This is not recommended.

Compiling the servo program

The servo program is compiled from 6 C files (servo.c, geomet.c, shell.c, servoUtil.c, servoShow.c, and checkSun.c) which are in the cactus/tracker/ directory. See the Makefile in that directory. So, simply typing "make servo" will do the necessary compilation and linking to produce an executable "servo" file.

There is one further step required, though. This step changes the ownership of the servo executable file so that the program starts up as root. It needs to do this so that it can do the necessary system calls to change its Linux scheduler so that it runs with priority scheduling (SCH_FIFO) rather than the default Linux scheduling. This gives servo higher priority and allows it to have control of the processor a larger fraction of the time. It makes it more real-time. The servo program will run if this step is skipped, but it will eventually (on a time scale of a day or so) neglect the flow of data from the DSP processor for so long (2 sec.) that the program will halt.

To adjust the privileges on servo so that it can change its scheduler successfully, after the new servo file is created (by "make servo"), the programmer has to become root. Then run a script in the tracker/ directory called "servoPriority". This script will have no effect if the user who invokes it is not root. Remember to do a ^D afterwards to stop being root.

Issuing commands to the servo program

The servo program can be controlled by the operator manually either from a socket message or from an xterm window. It can be controlled from other programs via socket messages which they can send to it. The program incorporates a simple command parser and it keeps looking for lines of ASCII text input for the parser to process. There are around 50 different commands. The program now (after July 2002) watches both sources of input (stdin and sockets) at all times.

The command parser uses reverse polish syntax. An input line consists of command names and/or numeric arguments delimited by blanks. It starts at the left end of the line and parses out "words". A word is either a literal integer number, a predefined command name, or a mistake. Those are the only 3 possibilities. If the word is interpretable as an integer number, the equivalent binary number is computed from the ASCII text and that value is pushed on the parser's parameter stack. If the word is a command name matching one of those in the command name list, then the associated command function is executed. All commands that take arguments, expect to take those arguments off the parameter stack. This is what enforces the reverse polish syntax, that is, the arguments must precede the command name on the line. One feature that devolves from this is that multiple commands can be typed in the same line. It's an especially simple way to parse commands.

For example, the command to print out some diagnostic status information is called 'st'. To invoke it, just type those two characters followed by a carriage return. st takes no arguments.

0-> st

The "0->" is the prompt printed by the servo program. The digit, '0' in this case, is a count of the number of numbers on the stack at the moment. The prompt is visible only from the shell. Programs or users communicating with it via sockets will never see a prompt.

As another example, the 'track' command takes 2 arguments. They are flags for azimuth and elevation drives. The command

0-> 1 1 track

tells it to start tracking in both azimuth and altitude. '0 0 track' would halt tracking in both axes. The first argument is the flag for azimuth and the second is for elevation. So '0 1 track' would disable tracking for azimuth and enable it for elevation. Note the blanks separating the literal numbers. The commands are listed in an array in shell.c called 'commands'. They are summarized in appendix 1 of this document.

Report processing

Normally the servo program is hung in a function called getReport() pending on a read from the DSP. The DSP sends a position report every 10ms (nominally, this can be changed dynamically). This is a long time compared to the time it takes servo to remove a report from the hardware FIFO and act on it, so usually the hardware FIFO between the DSP and Linux is empty. When servo does a read from the hep2e driver (via a call to api_read()) and the FIFO is empty, the call hangs until the DSP writes some report to its comport. That should not be more than 10ms. servo always sets an alarm prior to each read from the hep2e driver so that if the read hangs for more than 2 seconds without receiving a report from the DSP, Linux calls servo's timeOut() function which executes a shutdown routine to stop the telescope.

When the tracker task wants to send a new polynomial to servo, it does so by doing a special ioctl call to the IRIG driver. In this call it supplies the two 2nd order polynomials which specify the azimuth and elevation track as a function of time which the telescope should follow for the next 10 seconds. The driver saves this information in its local memory. The next time the IRIG driver receives a read command specifying a read length equal to that of the struct COEF_CMD (as opposed to the usual struct TS_TIME), it includes the latest polynomials in that COEF_CMD structure along with the current time. Reads of this type happen at least every 10 ms. If the read is the first one after a new polynomial has been specified by tracker, tracker will have set the 'new' flag in that structure to 1 indicating that it has an updated polynomial. If it is also for a new position (i.e., change of object or beam switch) then the 'new' flag is set to 2 (this is done in sendservo() in traksub.c). This is true for only that one read; on very next read (10ms later) with that same length, the IRIG driver will have the 'new' flag set to 0 again. This scheme is a hold over from the 12 meter telescope's control system (except for servo.c most of the SMT control software is taken from the 12 meter telescope). Note that because of this behavior of the IRIG driver, no other process should do IRIG reads with length equal to that of the COEF_CMD structure. That would cause servo to sometimes fail to recognize new, updated polynomials. All other calls to read the IRIG clock should use the TS_TIME structure defined in irigSmt.h.

As soon as getReport() has a new report from the DSP it does a read to the IRIG device driver to get the Universal Time (UT). It stores this in the global variable utTime. (The units of time are always milliseconds.) So, utTime is always the arrival time of the last received report of any kind. When getReport does this call to the IRIG driver, it uses a data length equal to that of the structure COEF_CMD as described above. If getReport() sees that the "new" flag in the structure is nonzero, it copies the whole irigTimeCoef structure to the "commanded" structure. So, normally, the 'commanded'

structure always contains a copy of the current position polynomials. If it sees that 'new' is = 2, then it also sets the 'newPosition' flags in the azSpeed and elSpeed structures. This alerts the servo logic in the zeroIn() function that a move to a new object is required, so it should switch from tracking to slewing.

When servo receives any new report from the DSP it uses a large switch statement (in the mainLoop() function) to test the report type code in the first 32-bit word of the report. Each type of command has its own case in the switch statement. By far most of the reports received from the DSP are position reports. Additionally about 20 "time reports" per second are thrown in. The time report is a no-op that just says what UT time the DSP thinks it is. It is used as sort of a software watchdog. If servo sees the position reports stop coming and the time reports don't stop, then servo stops the telescope. This can happen, for instance, if the DSP processor stops getting timer interrupts. The DSP's interrupt clock is derived from the IRIG time interface, so a failure in that or the cable connecting the 2KHz signal from the IRIG to the DSP counter/timer can cause this to happen.

Most of the other types of reports that can come from the DSP are not sent by the DSP unless servo specifically asks for them. For instance, servo will receive a corrections report (describing the ellipse fit parameters) only if it sends a command to the DSP asking for one. (It does ask for them every 10 seconds so it can report if any of the correction coefficients start looking wrong. See nextCorrCheck and checkCorrections().)

The only other types of reports that the DSP sends unsolicited are reference pulse reports. Those are sent whenever a reference pulse is seen on any of the encoders. The reference pulse is not actually used to alter the DSP's cycle count unless the DSP has previously been "armed" to do so on the next ref. pulse AND the pulse is seen while the telescope is going in the forward direction. But, regardless of whether the DSP uses the reference pulse event or not, or in which direction the telescope is moving, all reference pulse events are reported to servo by the DSP. The reference pulse report indicates if the cycle counter was actually used or not. See the section below about the reset-encoder operation which is the principal use of the reference pulse.

There are a number of diagnostic reports which can be sent by the DSP code. These are not used in normal operation. While debugging the DSP code, they were inserted into the code at various places to see what the DSP was doing. The DIAGNOSTIC4 report, for instance, just sends 4 32-bit numbers to the Linux host, with a timestamp. When servo receives a diagnostic report, it just prints out its contents. Except for the timestamp, the contents of the diagnostic reports have no predefined meaning. They are not used in the normal operation of the program.

Processing position reports

When servo receives a valid position report, it calls a function called positionServo(). This does all the processing necessary to decode the position and to update the velocity signals that go out to the Krupp rack to adjust the telescope position to keep it on track. It closes the loop. It has been measured taking 23 microseconds to do this, each time it receives a new position report from the DSP. Compiling servo with different optimization options will probably affect the performance. The 23 microsec. figure was obtained with optimization set to level 2. The standard makefile in tracker/ compiles for the Gnu debugger which will probably degrade performance a little.

positionServo() will stop the telescope if the position polynomials that it is using get to be more than 60 seconds old. They should normally get updated every 10 seconds. If they start flowing again, the telescope will resume tracking. (If it has to move a large distance to catch up, it probably won't do so very gracefully, but subsequent motions should be normal.)

positionServo() also increments a count every time it receives a position report. It outputs bit 5 (=64) of this count to the Krupp rack as the watchdog signal. That oscillates at about 1.5 Hz. If this ever stops changing for as long as 2 seconds, the Krupp rack stops the telescope regardless of what the computer is telling it.

The SMT has only one Azimuth encoder, but it has 2 elevation encoders. The servo program is dynamically configurable to use either or both of the elevation encoders. Normally it uses both and takes as the elevation position the average of the readings from the 2 encoders. If the difference between the 2 encoders gets to be too large (< 0 or > 55 arcsec.), it outputs an error message and picks one of the 2 encoders to use and ignores the other. The encoder that it uses is determined by the global variable 'elevFailSafe'. It is set equal to encoder 1 by default. It can be changed to encoder 0, by editing servo.c and recompiling. Note that servo refers to the two elevation encoders as 0 and 1. Much of the SMT engineering documentation refers to them as 1 and 2.

You can manually enable or disable the use of either encoder by using the 'el0' and 'el1' commands. Each takes an argument of 0 or 1 to disable or enable the associated encoder. See appendix 1.

positionServo() does a few error checks and computes the azimuth and elevation as a floating point number of arcseconds.. It then calls closeLoop(). closeLoop() evaluates the current position polynomials to calculate the desired azimuth and elevation and a desired speed at the current UT. These are checked to make sure they are within range and then passed to the zeroIn() function. If the system is tracking (i.e., not acquiring a new object or position) zeroIn() will just in turn call the voltage() function which handles tracking. If a new object has been selected though, zeroIn() takes over and handles the slew to the new position. When the telescope gets to the target position, zeroIn() reverts to it's normal tracking behavior which is to just call voltage() to have it do the fine tracking.

The voltage() function has the logic for the basic servo tracking response of the system. The expressions and constants used in voltage() are exactly the same as those used in the original VAX-based control system for the SMT. voltage() computes the current pointing error angle by subtracting the current position (from the DSP) from the target position (from the polynomial given the current UT). Then it uses the servo transfer function to compute the velocity which the telescope should have at the current time. It then calls commandVelocity() with the desired velocity in arcsec/sec. commandVelocity() converts that to an equivalent unsigned integer value to be written to the 16-bit D/A converters (one for azimuth and one for elevation). This returns the value to voltage() and also outputs it to the control system via the 16-bit D/A converters.

Voltage() also can handle slewing, but not very gracefully - it has a tendency to overshoot a bit. That's why zeroIn() normally intercepts big position changes before voltage() ever sees them and handles them first. Should voltage() ever be called with large position errors (=many arc minutes or degrees), it will do its best and get there eventually.

Finally, after all that computation, voltage() checks two global flags for each axis called driveRequest and driveEnable. The driveRequest flag is set when the 'track' command (see appendix 1) is used to command the program to attempt to drive the telescope in one or both axes. They flag that the operator is requesting servo to drive the telescope. The driveEnable flags are validated by a function called checkEnables() which is executed every time any report is received from the DSP processor. checkEnables() examines all of the 48 input bits connected to parallel ports 2 and 3. These contain signals from the Krupp rack which indicate unsafe conditions and system status, and also whether the Krupp rack is accepting computer control commands. If these status bits indicate all is well,

checkEnables() sets the driveEnable flag for one or both encoders. If there is any problem, one or both driveEnable flags will be zeroed.

If either of the driveEnable and driveRequest flags is zero, the voltage() function then forces the desired velocity signal to 0, and stops the telescope from moving. (This actually starts a controlled deceleration to 0 velocity which may take several seconds if the telescope is moving fast.) So, if, for instance, the elevation cabin door is opened, the telescope will stop. However, servo will continue to compute the desired position as a function of time and continue watching the telescope's actual position. As soon as the inhibiting condition goes away - some one closes the door - it resumes tracking. Since the position error will, in general, be large by that time, the servo transfer function will compute a large error correction signal. That is, the telescope will attempt to get back on track as quickly as possible by accelerating the telescope to the proper position and then resuming normal tracking.

Each time a new velocity is computed, the corresponding voltage signal is compared to the last commanded voltage. The new commanded voltage is not allowed to change by more than MAX_VOLT16_CHANGE from the last one. Nominally this is 21.84 counts of the D/A interface - enough to accelerate at 0.1 deg/sec/sec. This is fast enough to bring the telescope from max. speed (nominally 4000 arcsec/sec) to a stop in a few seconds.

Any and all functions that stop or drive the telescope using this same voltage() function, so all speed changes are subject to this MAX_VOLT16_CHANGE acceleration/deceleration limit.

Resetting the Encoders

The shaft encoders are relative encoders, producing one full sine/cosine wave output for every 36 arseconds of shaft rotation. But they also produce a reference pulse signal as the shaft turns through a particular absolute angle position. The reference is a signal that marks a particular sine wave. The elevation encoders seem to have only one reference pulse at about 23 deg., but the azimuth encoder has a ref. pulse every 45 deg. of rotation. Unfortunately, most are not clean looking single pulses, but multi-peaked, wide, and generally very irregular. Their irregularity is quite repeatable, though. We have chosen to use the reference pulse at 90 degrees azimuth for that axis.

Observations with the telescope have fixed absolute angle values to the sine waves that are marked with the reference pulse. For the 3 encoders these fiducial positions are:

elev. 0: 2295 waves + 10.0 arcsec = 82630 arcsec = 22deg. 57 min. 10sec.

elev. 1: 2394 waves + 4.5 arcsec = 23deg. 56min 28.5 sec.

Azimuth: 8903 waves + 18.0 arcsec = 89 deg. 2 min. 6 sec.

So, for example, when the azimuth ref. pulse is detected, the system can reset the azimuth wave counter to 8903 plus 18 arseconds. All subsequent angle motion is measured relative to that position. Of course the encoder reading is not exactly the telescope pointing direction. Because of various sources of flexure and/or axis misalignment, small corrections still need to be applied to the encoder readings to determine the true telescope position. The telescope "pointing model" is used to do that. It is determined empirically by pointing the telescope at various known objects distributed across the sky and noting the small deviations between the encoder readings and the actual position on the sky. These deviations are then least squares fit to a functional form which is chosen to represent telescope flexure and atmospheric refraction. This document does not concern itself with anything beyond the absolute encoder readings, however.

The process of locating the encoder reference pulses and setting the absolute encoder reading is called "resetting the encoders." It is done by moving the telescope to a position just before the position of the reference pulse and moving the telescope forward (towards increasing angle) slowly until the reference pulse is seen by the DSP. The DSP (if it has been "armed" to do so) sets its cycle counter to the a priori value listed above as soon as it sees the reference pulse. It also sends a reference pulse report to the Linux host. In principle the positions output by the DSP thereafter are absolute and accurate to about 1 arsecond (provided the pointing model corrections mentioned above are performed).

Until the DSP has seen the reference pulse (or otherwise been commanded by servo to set its cycle counters to a specific value), the position data that it sends to the Linux processor is completely inaccurate.

The telescope also has "coarse encoders" on each axis. These are just electrical potentiometers whose resistance is linearly related to shaft position, so they can be used as absolute shaft encoders, albeit not very accurate ones. Within a few seconds of starting the first motion of the telescope (as soon as the DSP indicates it's done an ellipse fit), servo reads the coarse shaft encoders (via a 12-bit A/D interface) and that position data is sent to the DSP so it can set its cycle counters. These coarse positions are only accurate to a degree or so. They are sufficient to be able to position the telescope to a point before the reference pulses and then do the procedure described above to reset the encoders.

Any time the DSP sees a reference pulse signal, it reports it to servo which logs it to the Monitor task. A reference pulse report tells which encoder generated the pulse, whether the telescope was moving in the positive or negative going direction on that axis when the pulse happened and whether the reference pulse was used to alter the encoder's cycle count. The cycle counter will not be altered unless servo has previously "armed" the DSP's ref. pulse logic to do so. If it was armed, the logic disarms itself automatically after the cycle counter is changed, so that a subsequent ref. pulse event, while it will be reported, will not change the cycle counter. The ref. pulse report also shows the value of the cycle counter when the pulse was sensed regardless of whether it was changed. After the encoders have been reset, then every time the reference pulse is passed in the positive going direction, it should always print out the value of the a priori cycle count listed above. If, after the encoder is reset, that number does not repeat, then somehow the DSP lost count of the encoder cycles, or the encoder itself has malfunctioned. The reference pulse report doesn't print the value that the counter is changed to (if it was changed), because that is always the same.

Resetting the Encoders Manually

The encoders can be reset manually or automatically. To do it manually, just send the servo program a command to arm the reference pulse logic for each axis. This is done with the commands "azrefmode" and "elrefmode". They take no arguments. After these are executed, the next time the DSP sees the reference pulse when it is moving in a positive direction (toward increasing angle), it will reset its cycle counter for the armed axis. It doesn't matter if the servo program is driving the telescope when this happens or if the operator is moving it in manual mode. You can do this before servo has been told to start driving (via the 'track' command).

So, after giving servo the arming command, manually move the telescope to a position before the reference pulse fiducial and slowly drive it forward until it reports seeing the reference pulse. You can then back up over the ref. pulse and, without re-arming it, run the telescope forward over the fiducial again. Any time this is done after the first armed encounter of the ref. pulse, the ref. pulse report should show the same cycle count.

Resetting the Encoders Automatically

The automatic encoder reset operation uses the servo program to do all necessary arming and telescope motion automatically. The telescope must first be moving under computer control, of course.

The encoders are reset automatically by sending the servo program the commands "resetaz", "resetel", or "resetboth". Each axis can be done separately or both can be done simultaneously. It moves the telescope to a position before the reference pulse, arms the ref. pulse logic, and then moves it forward at 1/10 slew speed until it sees the reference. It will report the reference pulse whenever it is seen. It then automatically backs up over the reference position and runs through it twice more without arming it so the observer can see if the results, that is, the cycle counts are repeatable.

Note that since the SMT has such a large solar avoidance zone, there may be long periods (hours) when the point on the sky which represents the fiducial indexes for both axes is within the zone of avoidance. (That's the point at az=90deg, el=23 deg.) In that case, using the 'resetboth' command will drive the telescope too close to the sun, so it will reject the command. When this happens, you have to use the 'resetaz' and 'resetel' commands separately. Move the telescope to a point above or below the solar avoidance zone and execute 'resetaz'. When it has finished that, then move it to a point left or right of the avoidance zone and execute 'resetel'.

Hiding from the Sun

The SMT's 10 meter dish is capable of concentrating destructive levels of solar energy on the receiver, the secondary, or its supports if it is pointed too close to the sun. For this reason the telescope has always been operated with a 30 degree solar avoidance zone. The software will ignore a command to acquire an object that is within 30 deg. of the sun. (That will in fact cause it to pause, so it will require the operator to unpaue it to get things going again.) If it should ever find itself within the solar avoidance zone, the software attempts to move the telescope out of it as quickly as possible.

The mainLoop() function normally checks for solar proximity every few seconds. The software will tolerate a "grazing" incursion into the solar avoidance zone under some circumstances. If it detects the telescope is less than one degree within the zone it doesn't automatically start a hide-from-sun operation. It shrinks it's sun checking time period to about 1/10 sec. and keeps watching it. If the telescope stays within the outer 1 degree of the zone for more than about 10 sec., the software takes over and hides from the sun. If the incursion ever exceeds 1 degree (i.e., < 29 deg. from the sun) it will hide from the sun immediately. This behavior is to allow a minor, brief incursion to be tolerated.

When the software decides to hide from the sun, it first computes the location that point which is just outside the solar avoidance zone which it can reach the fastest, given its present position and velocity. It then goes to that position using the "direct" slew method described below. This is designed to minimize the time the telescope spends inside the avoidance zone. Once it is at this nearby safe point, it computes a more distant safe point which is 20 or 30 degrees away and goes to that point and stops. Generally this will be at a numerically lower azimuth than the sun so that the sun will then be moving away from the telescope.

If one axis is disabled, the program does the best it can to get to a safe spot. It is possible in this case for the program to get caught in a "pinch point". For example, if the telescope is very low in elevation and the solar avoidance zone is just above it and moving lower and the azimuth axis is disabled, it can get stuck. It could move above the avoidance zone in elevation, but not without passing through the avoidance zone just above it. In this case the program looks at how close it would get to the sun if it

were to run up to a high elevation. If that distance is greater than 19 degrees (see TOAST_LIMIT in servo.h to change this), it will go anyway. It requires the azimuth speed to be < 100 arcsec/sec before it will chance this. If the > 19 deg. condition can't be met, or the azimuth axis is moving faster than 100 arcsec/sec, it will move down to elevation = 2 deg. and just keep watching the situation. Similar logic is used for the case where the elevation axis is disabled and the azimuth is drivable.

Sun avoidance: Slewing around the sun

There are 2 slew algorithms which the servo program uses to go from the current position to a new position. If the operator clicks on the "Force Direct" button in the servo control panel, the "direct" algorithm is used for the next slew only. Otherwise it uses a more complex "sun avoidance" algorithm which computes a path that is guaranteed to keep the telescope at least 30 deg. from the sun at all times. If a straight line path between the start and end position does not encroach on the sun avoidance zone, that straight line path is followed. If it does cross the avoidance zone, then a path that curves around the zone is used instead.

The direct algorithm runs the azimuth and elevation axes completely independently. It computes the total change in AZ and EL needed to reach the target location and at arrive there at approximately the right target velocity. Each axis is ramped up in speed to the maximum (nominally 4000 arcsec/sec, although it can actually go as fast as 5400 arcsec/sec), held at that maximum for a precalculated period, and then ramped down so that it reaches the target value at the desired velocity. If the distance is short enough, it may not be able to reach terminal velocity before having to start decelerating. So, a graph of velocity as a function of time for one axis is either going to look like a triangle for smaller movements or a trapezoid for long ones. Movements greater than about 11 deg. produce the trapezoidal case. The axis that has the smaller distance to go will finish slewing first and start tracking while the other axis is still slewing. There is no coordination of movement between the 2 axes.

The sun avoidance path algorithm computes a list of AZ EL positions, at roughly 1 degree increments (1 deg. on the sky), which define a path from the start point to the target. The path is guaranteed to always stay at least 30 degrees from the sun.

First the program tests a "straight" line in AZ-EL space (which is not a straight line on the sky) from the start point to the finish point. It checks this line at one degree intervals to see if it stays clear of the sun. (Computing the distance to the sun is just the dot product of the unit vector to the sun and the unit vector to the point on the path, so it is computationally cheap - no trig functions involved except for the very first point.) If all points are far enough from the sun, the straight path is used.

If the straight line in AZ_EL space crosses the solar avoidance zone, it uses a "stretched string" algorithm to compute a path around the sun. First the 30 degree radius circle on the celestial sphere centered on the sun is expressed as a list of about 200 AZ-EL points. In the AZ-EL coordinate system it is a distorted circle. From this point on, the problem is two dimensional in AZ-EL space. The path is what you would get if you stretched a string from the start point to the end point, but deflected it around the sun avoidance zone, as though it were a physical barrier. This is accomplished by computing a straight line from the start point to a point tangent to a slightly expanded sun avoidance zone and the other tangent line from the destination point to another tangent point to the sun avoidance zone. Between the 2 tangent points, the (slightly expanded) sun avoidance zone perimeter is taken as the path to be followed. By piecing these 3 segments together, a list of points from the start to the destination is compiled at about 1 degree intervals.

The program then computes times, velocities and accelerations for the start of each interval in the list. The servo loop does its best to keep the telescope on this schedule to get it to the destination. It always accelerates along the arc of the path at 360 arcsec/sec to a maximum of 4000 arcsec/sec. (in one coordinate - see below). When it gets to just the right spot near the end of the track it decelerates at 360 arcsec/sec along the arc so that it ends up at 0 arcsec/sec. at the target.

When following a sun avoidance path, velocities are chosen so that one of the axes is always being driven at maximum speed (assuming the path is long enough to allow reaching maximum speed) of 4000 arcsec/sec. As the path curves around, that may switch seamlessly from one axis to the other, but one of them will always be running at 4000 arcsec/sec.

If the operator gives servo a new target position (or a "halt" command) before it finishes the slew path it is on, it computes a new series of accelerations and velocities along the same path it is following that will bring it to a halt. This normally takes about 5 or 6 intervals (5 or 6 degrees). Once stopped it then computes a new path to the new object and takes off for that new target. (In the case of a halt command, it actually back tracks a bit to the position it had when the halt command was given and stops there.)

The cable wrap

The SMT can rotate a total of 540 degrees in azimuth. Azimuth is measured from -90 degrees at due West around clockwise to 450 degrees (due East). In reality the program de-rates these limits by 2 degrees, so the usable range is from -88 to + 448.

The servo control logic defines the "positive wrap" as the AZ values in the range 272 to 448 degrees. The "negative wrap" is the AZ values in the range +88 to -88 degrees. These regions overlap. So an object at AZ=10 degrees can also be accessed at AZ = 10 + 360 = 370 degrees. The servo logic normally takes the shortest path when moving from its present position to a new source. If it is at AZ = 10 deg. and is commanded to a source at AZ = 355, the shortest path is to go to AZ = -5 deg., so it will move counterclockwise to that position. However, the operator may want to force it to wind around the long way. You can force it to use one wrap or another by using the "usewrap" command.

"usewrap" takes one argument which is 1, 0 or -1. A 1 tells the program to always use the positive wrap, 0 tells it to go the shortest way (this is the default) and -1 tells it to use the negative wrap. So, if an object is at AZ = 10 degrees and you have previously executed "1 usewrap", servo will find this object to be at 370 because that is in the positive wrap. After you call usewrap, the wrap that you specify remains in force until you specify otherwise by another call to usewrap. The default is 0, that is to use both wraps, whichever is closer.

Exceeding the Lower Elevation Limit

The normal lower elevation limit is 2.0 deg. (see EL_MIN_LIMIT_DEG_DEFAULT in servo.h). Normally the computer will choose paths that are always above this value. If an object moves below this limit, the telescope will not follow it, although it continues tracking in azimuth.

Two degrees is the elevation at which the "Elev. Pre-limit down" signal from the Krupp rack turns on. When this happens the Krupp servos automatically limit the elevation speed to 10% of maximum. If the telescope is to move below the 2.0 deg. limit, the computer has to set its own Elevation speed limit to 10% if it is to be able drive the telescope properly in this region. They both have to use the same rules.

To exceed the 2.0 deg limit the operator has to do 2 things. First he has to set a new minimum elevation limit. This is done with the "elminsec" command. It takes an argument which is an integer number of arcseconds. So, "3600 elminsec" sets it to 1 degree. "0 elminsec" sets it to zero degrees. Next the operator has to set the maximum velocity allowed in this region. This is done with the "maxvelsec" command. Its integer argument is in units of arcsec/sec, so "400 maxvelsec" would set it to the 10% value that the the Krupp rack enforces below the 2.0 degree pre-limit. (Note that this can also be used to set the velocity limit to values higher than the 4000 default value. Not recommended, though.) After both of these steps have been done, the program will respond to commands to move to positions with elevation < 2.0 deg. There is a hard limit at 0 degrees. Below this, the Krupp rack ignores the computer's velocity signal.

When you are ready to move the telescope back above elevation 2 degrees, you can command it to a position at or just above 2 degrees (it's running at only 10% of max speed, so don't tell it to go a long way), or just move it manually above 2 degrees and then use the 2 commands described above to set it back to its normal condition.

Appendix 1

servo commands

The servo program has a simple command parser and looks for text input from either the keyboard or a socket at any time. The syntax for the ASCII commands is the same regardless of how they are transmitted to the program. After each command is executed, a response is sent back to the user. It starts with the ASCII characters "OK:" or "FAIL:" followed by some text which is peculiar to each command which confirms what happened. Many commands can only return the "OK:" response. See the individual commands in servo.c to see what is returned as a response for each command.

Note that most of the commands below are for diagnostic use and won't make much sense to the telescope operator. All commands are documented, but only a small portion of them are useful to the operator.

In the lines below showing the call sequence for each command, the (usually) right most word printed in bold is the command name. The words to the left of it (if any) are the arguments that it takes. They are to be specified as numeric literals. Any words that may be shown after it are numbers the command leaves on the parameter stack. More than one command can be placed in one line of text. Blanks are delimiters. For example in the example below the command FOO takes 2 arguments, "a" and "b" and leaves a result "c".

a b **FOO** c

An example of a function that does this is "+". It takes 2 arguments, adds them and leaves the result on the stack. so "1 5 +" leaves a 6 on the stack. Hardly any of the commands leave a result.

Note that the names are case sensitive!

abort: Aborts a resetboth, resetaz or resetel command that is in progress.

abort

Any reset encoder operation that was started by a 'resetaz' 'resetel' or 'resetboth' command is immediately aborted by this command.

<takes no arguments>

AZ: specify a particular channel to be one of the Azimuth signals

group chan sig **AZ**

Assign one of the azimuth encoder signals to a particular channel number on a particular eight channel A/D. "group" identifies the A/D board - either 1 or 5 for the one connected to comport 1 or comport 5. "chan" is the channel number, 0 - 7. "sig" is the signal, SIN, COS or REF. Normally this is invoked from the servoCfg.txt file when it is loaded to declare the channel assignment. An example would be "1 0 SIN AZ" which declares the Azimuth encoder sine signal to be connected to group one, channel 0. This change will not take effect until sendchan is executed. See EL0, sendchan and showchan.

group group number = 1 or 5 to identify the A/D
chan channel on that A/D. Always 0 - 7.
sig Signal identifier. Must be SIN, COS, or REF.

azrefmode: Arm DSP to use the Azimuth reference pulse.

azrefmode

This "arms" the azimuth reference pulse logic in the DSP computer. After executing this, the next time the DSP sees the azimuth encoder reference pulse when the telescope is moving in the direction of increasing azimuth ("positive-going"), it will reset the azimuth cycle counter to the standard a priori value for that axis. This can be un-done, that is, un-armed, by using the mode0 command. That value is fixed in the general.h file. See elrefmode. Normally the encoder cycle count is reset in a more automatic way by using the "resetaz" or "resetboth" commands.

<takes no arguments>

c: Request a correction report and print it

c

This takes no arguments. It sends a command to the DSP that tells it to send back the current correction constants. The report specifies the offsets, amplitude and phase corrections for all 3 encoders. When any correction report is received, the correction numbers are checked to see that they are reasonable. If they are too far off, servo prints a warning message.

<takes no arguments>

co: Force read of coarse encoders and set az and EL cycle counters.

co

Forces a read of the coarse encoders and then does a setCycleCtr command to have the DSP set its cycle counters to the values just read in. This is largely diagnostic. It was used at a time when the coarse encoder reading was failing often due to high noise levels. Note that executing this after a reset-encoders operation will de-initialize the position, that is, reduce the position accuracy to that of the coarse encoders. It will be necessary to do another encoder reset to do accurate pointing again.

<takes no arguments>

coarse: Synthesize coarse elevation voltage

n coarse

This is a diagnostic function which takes an argument from 0 to 45. It uses an unused DAC channel to generate a voltage input to the A/D channel which normally carries the coarse elevation position. A short jumper cable connects the DAC to the A/D. This allows one to manually (before executing 'track') set a voltage on the coarse elevation input corresponding to an angle of 0 to 45 degrees. It can't do more than that because the DAC output is +-5v and the A/D input is +-10v. This was used to test the coarse encoder inputs. It also saves some time getting the test set up to the desired elevation position.

n Number of degrees corresponding to the desired coarse elev. input. Must be between 0 and 45.

COS: pushes a code number on the stack that identifies an encoder cosine signal.

COS n

COS is a "constant" function. It pushes a signal code number on the stack. See SIN and REF.

n the code number for a cosine signal.

dacset: Send a specified data value to a specified DAC channel.

ch d dacset

Sends a specified 12-bit data value to one of the 6 DAC channels. A data value of 0 produces -5v on the DAC output. A value of 4095 produces +5v on the DAC output.

ch The DAC channel number. Must be between 0 and 5.

d The 12-bit data value. 0 - 4095 produces -5v to +5v. 2048 gives 0v.

dasread: read all 16 channels of A/D and print raw readings

dasread

reads all 16 channels of the 12-bit A/D converter and prints out the raw data values. This is a passive operation. It has no effect on the telescope.

<takes no arguments>

dsperrors: prints a summary of DSP errors

dsperrors

Takes no arguments. This prints one line for each possible dsp error code and a count of how many times it has received each error message from the DSP.

<takes no arguments>

elrefmode: Arm DSP to use the Elevation reference pulse.

elrefmode

This is just like azrefmode, except it works for the elevation encoders. Affects both encoders. See azrefmode.

<takes no arguments>

EL0: specify a particular channel to be one of the Elevation 0 encoder signals

group chan sig EL0

Assign one of the elev. 0 encoder signals to a particular channel number on a particular eight channel A/D. "group" identifies the A/D board - either 1 or 5 for the one connected to comport 1 or comport 5. "chan" is the channel number, 0 - 7. "sig" is the signal, SIN, COS or REF. Normally this is invoked from the servoCfg.txt file when it is loaded to declare the channel assignment. An example would be "5 0 SIN EL0" which declares the elev. 0 encoder sine signal to be connected to group five, channel 0. This change will not take effect until sendchan is executed. See EL1, sendchan and showchan.

group	group number = 1 or 5 to identify the A/D
chan	channel on that A/D. Always 0 - 7.
sig	Signal identifier. Must be SIN, COS, or REF.

el0: Enable or disable the use of elevation encoder 0.

n el0

Takes a flag which indicates if elevation encoder 0 should be used or not. If n is 0, elevation encoder 0 is ignored. If it is nonzero, it is used. See el1.

n	Flag indicating if elevation encoder 0 should be used. 0=ignore el0. nonzero = use el0.
---	---

EL1: specify a particular channel to be one of the Elevation 1 encoder signals

group chan sig EL1

Assign one of the elev. 1 encoder signals to a particular channel number on a particular eight channel A/D. "group" identifies the A/D board - either 1 or 5 for the one connected to comport 1 or comport 5. "chan" is the channel number, 0 - 7. "sig" is the signal, SIN, COS or REF. Normally this is invoked from the servoCfg.txt file when it is loaded to declare the channel assignment. An example would be "5 0 SIN EL1" which declares the elev. 1 encoder sine signal to be connected to group five, channel 0. This change will not take effect until sendchan is executed. See EL1, sendchan and showchan.

group	group number = 1 or 5 to identify the A/D
chan	channel on that A/D. Always 0 - 7.
sig	Signal identifier. Must be SIN, COS, or REF.

el1: Enable or disable the use of elevation encoder 1.

n el1

Does the same function as el0 except it does it for elevation encoder 1. See el0.

n Flag indicating if elevation encoder 1 should be used. 0=ignore el1. nonzero = use el1

go: Send a "go code" to the DSP

go

This sends a go code to the DSP. When the DSP first starts up, the DSP processor hangs waiting for the go code before it does anything else. The DSP always responds to a go code command with an "initialization report". This contains the DSP's value of UT, the report period it was last commanded to use and some information on comport 5 which is used for the A/D data.

<takes no arguments>

halt: Stop a slew and reset the slew state variables.

halt

This command is also available as the "HALT" button in the tracker/servo control panel. "halt" assumes the slew path-following software has gotten confused, so it doesn't use it (as suspend does). It forces the path-following state variables into their initial condition and stops both axes as fast as possible. It then commands the telescope to go to a position which is equal to the position it had when the halt command was given. If the telescope was slewing when halt was executed, this means it will backtrack a few degrees to get to the halt position. It will then stay there. It is still tracking, it's just tracking an object that isn't moving. So, if you command it to a new position, it should be able to move to it normally. The real purpose of the halt command is to re-initialize the path-following software. So, if you see it having trouble executing a slew properly (sometimes it gets hung), use halt.

hatches: Move the telescope to the hatches position

hatches

This command is also available as a button in the "Service Positions" control panel. This moves the telescope to the hatches position at AZ = 265 degrees. The preferred elevation for the hatches position is 88 degrees. (See MAX_ELEV_SLEW_DEG in general.h.) However, when the sun is up, the system may be forced to choose a lower elevation to avoid the sun. It will always take a position above the sun if it can. If it can't it will choose a position about 10 degrees ahead of the sun. If it has picked a position that will eventually be overrun by the sun avoidance zone, it will print a warning message to that effect.

<takes no arguments>

help: Prints the names of all commands and summary of their arguments.

help

Prints out the contents of the command list along with summaries of the arguments each command takes (if any).

<takes no arguments>

keep: Force main while loop variable

n keep

Sets the 'keep' variable controlling the main while loop in servo to the value n. This is a diagnostic command and should not be used during normal operation. The loop runs only as long as keep == 1.

n Value to which the 'keep' variable is set.

manlog: Make a manual log entry

manlog <string>

Allows you to enter 1 line of text into the servo.log file. If there is a servo related problem you can type (in the servo xterm window) the word manlog followed by whatever text. It takes all the text after manlog up to the carriage return and places it in the servo.log file with a time stamp. So it might look like this:

manlog telescope wont move when a new object is selected

mode0: Command DSP to set its mode variable to 0

mode0

The DSP has a mode variable which basically tells it whether it is supposed to send reports or not and if it is armed to use the next positive going reference pulse that it encounters. A value of 0 is the "normal" value which has it doing reports and not armed for reference pulses. This is a manual way of forcing it to have a zero value. It allows one to undo an "arm" command. See azrefmode and elrefmode.

<takes no arguments>

nreport: Change the DSP's position report period

n nreport

Specifies to the DSP processor how frequently it should send a position report. The default is 10 ms. This can be changed at any time. It shouldn't be any faster than 6ms.

n The desired report period in units of milliseconds.

printpos: start/stop printing position report

n printpos

If n is nonzero, servo begins printing position reports every 2 seconds. If it is 0 it stops printing reports. The delay period is controlled by PRINT_PERIOD in servo.h.

n: Logical flag (0 or 1) to indicate if printing should be started or stopped.

readcfg: Reads in the servoCfg.txt file.

readcfg

Looks for a file called servoCfg.txt and 'loads' it. That is, that file contains executable shell commands. This reads the file and executes those commands. If the file cannot be found, it looks for a file called servoCfg.master and copies it to servoCfg.txt. This file contains declarative commands that define the assignment of encoder output signals to A/D input signals.

<takes no arguments>

rec: To record raw encoder data

chan encoderNum time rec

Starts recording 2 channels of raw encoder data from the specified encoders in files on the linux system.

chan 0 means record sine & cosine. 1 means record cosine and ref. pulse

encoderNum the logicl OR of 1, 2, and/or 4. 1 means to record Azimuth, 2 means record elevation
encoder 0 and 4 means record elev. encoder 1. So 7 will record all encoders.

time length of time that data should be recorded in units of milliseconds. So 1000 = 1 second.

recpos: Record all position reports for a time period

n recpos

This command causes servo to start recording all position reports to a disk file starting immediately and continuing for the specified number of position reports. The positions are recorded in an ASCII text file called positions.txt. Each line is one report. An argument of 100 would record the next 100 position reports. At the nominal rate of 100/sec. that would take 1 second of time. One record consists of 13 columns of information. The fields are:

- 1- The time from the IRIG clock (in seconds) when then position report was recieved.
- 2- The UT that was put into the position report by the DSP. This is an integer number of milliseconds.
- 3- The Azimuth expressed as a floating pt. number of arcseconds.
- 4- Elevation encoder 0 expressed as a floating pt. number of arcseconds.
- 5- Elevation encoder 1 expressed as a floating pt. number of arcseconds.
- 6- Observed Azimuth speed in arcseconds/sec. in exponential format.
- 7- Observed Elevation 0 encoder speed in arcseconds/sec. in exponential format.
- 8- Observed Elevation 1 encoder speed in arcseconds/sec. in exponential format.
- 9- Projected (computed) azimuth speed in arcsec/sec.
- 10- Projected (computed) elevation speed in arsec/sec.
- 11- RMS deviation for AZ encoder. This is the RMS of $1-(\sin^2 + \cos^2)$, so it should be <0.02 .
- 12- RMS deviation for Elevation 0 encoder.
- 13- RMS deviation for Elevation 1 encoder.

n Number of position reports which are to be recorded to disk.

REF: pushes a code number on the stack that identifies an encoder reference pulse signal.

REF n

REF is a "constant" function. It pushes a signal code number on the stack. See COS and SIN.

n the code number for a reference pulse signal.

resetaz: Reset the Azimuth encoder only.

resetaz

This is the same as resetboth except that it resets only the azimuth encoder.

<takes no arguments>

resetboth: Reset both Azimuth and Elevation encoders.

resetboth

This commands servo to start a sequence of automatic telescope motions to calibrate, or reset, both azimuth and elevation encoders. It automatically moves both axes (using the coarse position info) to a couple degrees before the fiducial index points. It then arms both azimuth and elevation and runs the telescope forward in both axes until the reference pusle is seen. It then backs up over it and backs up for 10 more seconds and repeats the process twice. After this is done, the DSP knows the absolute position of the telescope (assuming it repeated OK). This is controlled by state driven logic and is fairly robust in dealing with problems. It prints out the cycle counters at the time the reference pulses were encountered so that the operator can see if it is repeatable. The 'resetabort' command will stop a reset encoder operation that is in progress.

<takes no arguments>

resetel: Reset the Elevation encoder only.

resetel

This is the same as resetboth except that it affects only the elevation encoders.

<takes no arguments>

savefast: save the last 12 minutes of internal operating variables (fast changing information)

savefast

This is a diagnostic command whose purpose is to save lots of internal program information to a file on disk. It should be executed when the program is exhibiting some unexplained failure. This file can later be examined to try to figure out what was happening in servo. As the program runs, it saves about 100 different internal variables in a 6Mb circular buffer every 10 milliseconds. At any given moment the buffer contains a record of the last 13 minutes of program operation. The savefast command writes the contents of this circular buffer to a file in the cactus/binData/ directory. The file name is saveMmmDD_n.bin. Where Mmm is the month name (as in Mar, or Oct), DD is the day of the month (03 or 31, for instance) and n is a single digit - 0, 1, 2, 3, or 4. Each time savefast is called, it increments the digit. If the digit gets to 5, it is reset to 0. So if you execute savefast more than 5 times in one day, the new file will overwrite the old file with the same name. You can avoid this by changing the old file name to something else that doesn't fit the "saveMmDD_n.bin" form. NOTE: typing the 'stop' command to stop servo automatically does a savefast operation (and a saveslow). See saveslow below. <takes no arguments>

saveslow: Save the last 12 hours of slow changing internal operating variables.

saveslow

Every time servo receives a new position vs time polynomial (nominally about every 10 seconds) from tracker it saves that polynomial and about 100 other slow changing variables in a 0.5Mb circular buffer. At any given moment, that represents data from about the last 12 hours of more of operation. This command dumps that buffer to a file on disk. Combined with the savefast data it gives a pretty complete picture of what has been going on. Like savefast described above, this is intended to be used when there is some sort of servo to record information that might help the programmer figure out what the problem was. It follows exactly the same file naming rules described for savefast, except the substring "fast" is replaced with the substring "slow". See savefast.

sendchan: Send the new channel assignment to the DSP.

sendchan

sends the new channel assignment to the DSP. It makes the DSP and servo start using the new assignment. In general you should stop the telescope (0 0 track) and then, change the signal assignment and move the signal cables around to reflect the new assignment. Then restart the drive operation (1 1 track).

<takes no arguments>

setaz: Sets the azimuth encoder cycle counter manually.

n setAz

Set the azimuth cycle counter so that it indicates n degrees. N must be an integer. This is generally only useful in the lab for debugging purposes.

n Number of degrees that the azimuth cycle counter should be set to.

setel: Sets the elev encoder cycles counter manually.

n setel

Set the elevation cycle counters so that they indicate n degrees. N must be an integer. This is generally only useful in the lab for debugging purposes.

n Number of degrees that the elev. cycle counter should be set to.

setfb: specify length of feedback averaging period .

n setfb

Specified the feedback period in units of position reports. If this is 10, then the average of 10 consecutive position report velocity errors will be used as the velocity error. The velocity error is only adjusted every 10 report periods. Legal values for n are from 1 to 200. A value of 1 means no averaging at all.

n Length of feedback averaging period in units of position report time intervals.

setlag: Specify servo lag parameter

n setlag

The servo error signal is computed by comparing the current observed velocity to the commanded velocity a short time ago. The lag parameter specifies how long ago. A value of 1 is the last commanded value. A value of 8 is the commanded value 8 position reports ago. see VEL_HISTORY and VEL_LAG in servo.h.

n lag parameter. normally 1 to 8.

setloop: Specify open or closed loop feedback for object acquisition

n setloop

This selects between 2 possible servo loops used by zeroIn(). This affects position changes only (slewing, basically), as opposed to tracking. If n = 0, servo uses an open loop algorithm. If is n nonzero then it uses a closed loop algorithm. Open loop means that the telescope velocity is commanded to be a certain velocity (plus a user specified constant offset - see setveloff) for a certain time, but no error is ever computed and the commanded velocity is not changed during the time period. The closed loop algorithm computes an error (the difference between pre-computed velocity and actual velocity), averages that error over a specified time period (typically 5 - 30 position reports) and then feeds that back to the commanded velocity to try to bring the actual velocity closer to the computed values.

setrelax: set the relaxation factor used in servo feedback loop

n setrelax

The error that is applied to the current projected velocity is multiplied by a relaxation factor first. Normally relaxation factors are in the range of 0 to 1. Zero effectively opens the feedback loop. A value of 1 applies the full observed error to the velocity (often this is a bit too much). Since the mini-shell can't parse floating point numbers, the relaxation factor is specified with an implicit multiplier of 10. So n = 4 means a relaxation factor of 0.4.

n relaxation factor times 10. This must be an integer.

setveloff: specify the velocity offsets to be used by the open loop servo algorithm

AZpg AZng ELpg ELng **setveloff**

Allows the operator to specify constant offsets which are applied to the computed velocities in the open loop servo method used for position changes (not tracking). There is a different offset (in units of arcsec/sec) for Azimuth positive going, Azimuth negative going, Elev. positive going and Elev. negative going. Note these are integers. See setloop.

AZpg The offset to be used when slewing AZ in positive direction (Increasing Az).

AZng The offset to be used when slewing AZ in negative direction (decreasing AZ).

ELpg The offset to be used when slewing EL in positive direction. (Increasing EL).

ELng The offset to be used when slewing EL in negative direction (decreasing EL).

showchan: print out the encoder signal vs. A/D input channel assignment

showchan

prints out the signal names that are assigned to each channel on the 16-bit A/Ds which are read by the DSP. These are all encoder signals. Each of the 3 shaft encoders has 3 output signals: sine, cosine and reference pulse. See sendchan

<takes no arguments>

showin: Reads and prints out the 48-bits of parallel inputs

showin

Reads all 48 bits of parallel inputs and prints them out with a line for each bit. The printout shows the state of each bit. An asterisk is printed next to those status bits which are in a state that prevents the telescope from moving. See showout.

<takes no arguments>

showout: Prints out the current state of the 24 output bits.

showout

Prints a separate line for each of the 24 parallel output bits. This is intended as a diagnostic.

<takes no arguments>

SIN: pushes a code number on the stack that identifies an encoder sine signal.

SIN n

SIN is a "constant" function. It pushes a signal code number on the stack. See COS and REF.

n the code number for a sine signal.

slew: Move the telescope in azimuth (usually a long distance)

n slew

n is taken as a signed number of degrees which should be added to current azimuth. The argument is optional. If an argument is specified, it tries to go that many degrees in azimuth. If that distance exceeds the wrap limit, it tries negating the sign of the argument. If no argument is specified, it picks +-175 degrees, which ever would leave the telescope closer to 180 degrees azimuth. It attempts to use elevation = 85 degrees for this move. If that elevation would leave it within 35 deg. of the elevation of the sun, it chooses the sun's elevation + or minus 35 degrees. This command can be given an argument larger than 360 degrees. It is often used to do an inclinometer run, See slowlo.

n signed number of degrees by which to change azimuth.

slewlo

n slewlo

n is taken as a signed number of degrees which should be added to azimuth. The argument is optional. It uses the same logic as the slew command. (see above) but it attempts to put the telescope low in elevation whereas slew attempts to put the telescope high in elevation. The favored elevation is 3 degrees. However, if 3 degrees is within 35 degrees of the sun's elevation, it chooses the sun's elevation + or - 35 degrees.

n signed number of degrees by which to change azimuth.

st: Print status information

st

This dumps a lot of status information about the program. It shows the number of position reports received since it started, position report period, a count of IRIG failures, various state variables and other variables. It is largely for diagnostic use and the detailed nature of its printout may change from time to time.

<takes no arguments>

stop: Stop the servo and DSP programs

stop

This sends a termination command to the DSP and then exits the servo program. That causes the shutDown() function to be executed. This ramps the telescope speed down to 0, puts the Krupp rack in a safe state, and sets all analog output voltages to 0.

<takes no arguments>

stopdsp: Sends a terminate command to DSP

stopdsp

Sends a terminate command to the DSP. This causes the servo program to stop shortly thereafter because the position reports stop coming. This is a diagnostic command.

<takes no arguments>

stopservo: Stop the servo program, but leaves the DSP running.

stopservo

Servo commands the DSP to stop issuing reports and then exits. This shuts down the telescope, but leaves the DSP code running.

<takes no arguments>

suspend: stop the telescope's motion

suspend

Commands the telescope to stop during a slew. It keeps the telescope on the track that it is presently following, but decelerates it to a stop. It is still tracking, but it is tracking a non-moving object. So, it can then be commanded to anew position. This is similar to the 'halt' command, but more trusting. It only works if the slew path-following logic is working. If you have reason to believe the path-following logic is not working, use 'halt' instead. It's a bit more heavy handed, but will get the job done.

<takes no arguments>

swapchan1: swap 2 A/D channels in the comport 1 A/D

p q swapchan1

Changes the assignment of A/D channels in the 8 channel A/D attached to comport 1 of the DSP chip. This is provided to allow the operator to move an encoder signal to a different input channel in the event that that channel becomes noisy or stops working. p and q are the 2 channels being interchanged. They must be numbers in the range 0 to 7. After changing the assignment by calls to swapchan1, you actually make it take effect by calling sendchan. That sends the new assignment to the DSP. See showchan to see how the channels are assigned.

p an integer (chann. number) from 0 - 7

q an integer (chann. number) from 0 - 7.

swapchan5: swap 2 A/D channels in the comport 5 A/D

p q swapchan5

Changes the assignment of A/D channels in the 8 channel A/D attached to comport 5 of the DSP chip. This is provided to allow the operator to move an encoder signal to a different input channel in the event that that channel becomes noisy or stops working. p and q are the 2 channels being interchanged. They must be numbers in the range 0 to 7.

p an integer (chann. number) from 0 - 7
q an integer (chann. number) from 0 - 7.

track: Enable or disable tracking in one or both axes

p q track

Specifies which axes are enabled for tracking. The arguments are 0 or nonzero to stop or start tracking. This only enables tracking. Tracking won't actually start unless the hardware indications show that it is safe to track. Also, tracker has to be supplying polynomials.

p flag indicating if Azimuth tracking should be enabled. 0 = disable, nonzero = enable
q flag indicating if Elevation tracking should be enabled. 0 = disable, nonzero = enable.

unpause: Clear the specified pause reason

n unpause

When the telescope is paused by a speed error or a stale polynomial, etc., it stops driving and prints a message explaining the (perhaps multiple) reasons for the pause. Each reason is printed with a code number. The operator can manually clear the pause condition by invoking unpause with that code number. If that is the only pause condition, the telescope will resume driving. If there are others, it will remain paused. If the condition still persists, the telescope may pause again.

n code number of the condition being un-paused.

UNUSED: pushes a code number on the stack that identifies an A/D UNUSED signal input.

UNUSED n

UNUSED is a "constant" function. It pushes a signal code number on the stack. See COS and REF.

n the code number for no signal at all.

usewrap: specifies which wrap the telescope should use.

n usewrap

n is 1, 0 or -1. An argument of 1 says that all subsequent moves are to use the positive wrap. That is, the region from AZ = 272 degrees to 448 deg. Normally when slewing to an object, the computer picks the shortest path. So if it was at AZ = 10 degrees and it was command to go to an object at AZ = 355 deg. it would go to AZ = -5. But, if wrap is set to 1, it will wind all the way around clockwise to AZ = 355. Similarly the neg wrap is the region from AZ = 90 to AZ = -88. If the wrap is set to -1, it will use the neg. wrap in favor of the positive wrap. Calling usewrap with an argument of 0, clears the wrap flag and the program follows its normal behavior of choosing the shortest path to an object.

1 use the positive wrap (270 to 448)
0 clear the wrap flag, go the shortest way
-1 use the negative wrap (-88 to +90)

ut: Set the time counter used by the DSP

ut

This command sends the Linux host machine's latest reading of the Universal Time to the DSP (in milliseconds) and has it set its time counter to that value. The DSP time stamps all reports that it sends to the Linux host with the UT. The Linux host and the DSP both use the IRIG clock as their time base, so they should not drift relative to each other. There is a small chance that in between reading the UT

and sending it to the DSP, the Linux host will be pre-empted by some other process. That would produce a fixed time difference between the Linux host and the DSP. For this reason, servo always monitors the difference between its UT value and the value of the time stamp in the reports from the DSP. If the difference is greater than 10ms for as long as a minute, servo repeats the "ut" command.
<takes no arguments>

wrapon: Enable movement into the + or - cable wrap or both.

n wrapon

If the argument $n = 0$, it re-enables movement into both wraps. If $n = 1$ it re-enables only the positive wrap (AZ = 360 to 450 degrees). If $n = -1$ it re-enables movement into the negative wrap (AZ = 0 to -90 deg.)

n -1, 0 or +1 to re-enable just the - wrap, both wraps or just the + wrap respectively.