CHARA Technical Report

No. 127 15 Aug 2024

Improved passive cooling of CHARA telescopes enabled by movable enclosure walls

James Wen[ab] , Nic Scott[a]

ABSTRACT: A new motion control system was designed and installed at the S2 and W1 telescope enclosures which will allow venting of hot air. This has resulted in measured improvements to telescope cooling performance as well as a slight improvement to seeing.

1. INTRODUCTION

The CHARA Array's six main telescopes enclosures consist of a dome supported by four steel columns, whose interior space is enclosed by two vertically stacked cylindrical walls. The upper cylinder has a wider diameter than the lower cylinder, and both are attached to a screw jack mechanism inside the support columns that allows them to move up or down to expose the enclosure to the outside environment. Their main function is to facilitate cooling of the telescope and onboard electronics by equilibrating the inside and outside temperatures. While the atmospheric conditions on Mount Wilson contribute to good seeing, the air inside the dome itself also contributes to the overall seeing of the telescope; this contribution is called "dome seeing". Minimizing the difference between inside and outside temperatures is known to improve dome seeing by reducing the convective mixing of hot and cold air[1].

The lower cylinder rests on the bottom platform of the enclosure when closed and moves upward to open, creating a small gap between it and the platform. When both the dome and the lower cylinder are open, warm air within the enclosure rises out through the dome slit and cool air is drawn in through the lower gap, leading to a chimney effect. The upper cylinder encloses the space between the lower cylinder and the dome. It moves downward to open, creating a gap between its top edge and the dome's bottom edge. This may reduce convective turbulence in the optical path by allowing hot air to escape from the sides rather than through the dome slit. It may also contribute to improved seeing by allowing wind from the outside to flow in a uniform sheet across the mirror. The height of the gap can also be adjusted in response to changes in humidity or intrusion of dust particles carried by the outside air.

The cylinders at all six telescopes have been inoperable for several years, leading to significant heating effects on the interior air, the deformable mirror (DM), and the adaptive optics board (AOB). We designed and implemented a new motion control system which restores cylinder movement at S2 and W1, allowing for improved cooling due to wind exposure. Preliminary results indicate that opening only the lower cylinder results in $\approx 1°C$ - $4°C$ reduction in temperature, as well as a mild improvement in seeing. Further testing over several months is required to make more confident conclusions about the cylinders' cooling performance. Additionally, the effect of opening the upper cylinder has not yet been tested. This technical report details the design and components of the motion control system, the hardware and software interfaces for onsite and remote operation, and the resulting effects on temperature and seeing as measured by various sensors.

[a]Center for High Angular Resolution Astronomy, Georgia State University, Atlanta GA 30303-3083
Tel: (404) 651-2932, FAX: (404) 651-1389, Anonymous ftp: chara.gsu.edu, WWW: http://chara.gsu.edu

[b]Department of Physics and Astronomy, University of Southern California, Los Angeles, CA 90089-1483

[1]N. Woolf, "Dome Seeing". *Publications of the Astronomical Society of the Pacific 91*, pp. 523-529, August 1979. Online access.

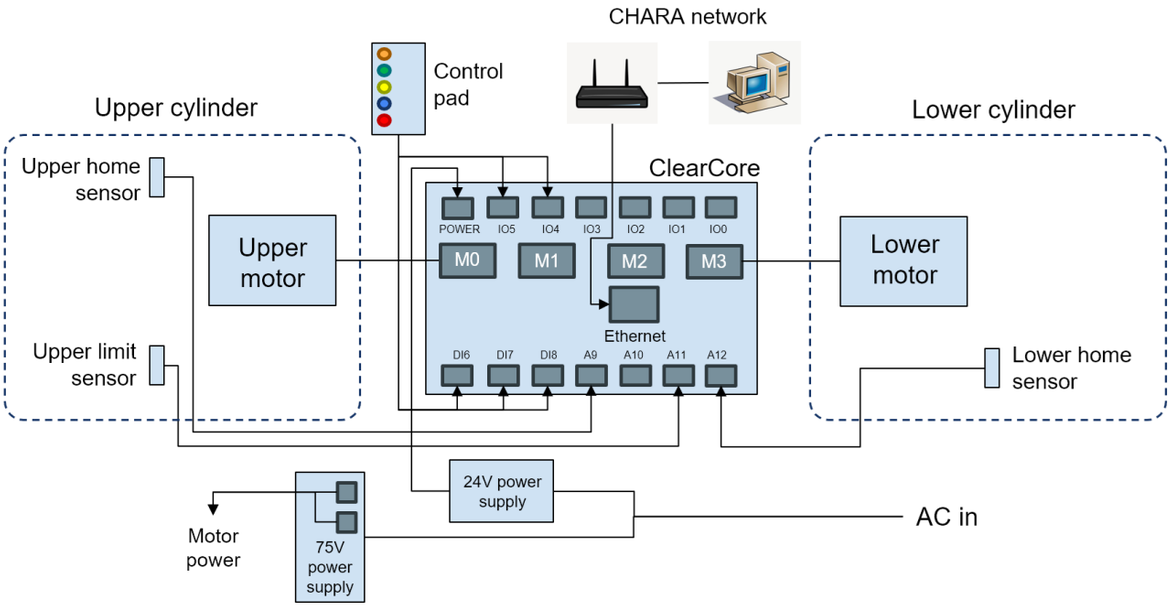

FIGURE 1. Block diagram of the new cylinder control system.

2. SYSTEM OVERVIEW

The cylinder control system as shown in Figure 1 consists of the following components:

- Teknic ClearPath-MCVC servo motor (manual)
- Teknic ClearPath-SDSK servo motor (manual)
- Teknic ClearCore I/O and motion controller (manual)
- Teknic IPC-5 75V DC power supply (manual)
- 24V DC power supply[2]
- 3 McMaster-Carr metallic-object proximity switches[3]
- 2 latching (toggle) buttons
- 3 momentary (push) buttons
- Newsuper waterproof enclosure box

2.1. Installation

The ClearCore, power supplies, and control pad are mounted inside the waterproof enclosure box, which is installed at the base of the telescope enclosure. The ClearPath-MCVC and ClearPath-SDSK motors are installed outside of the enclosure and mechanically linked to the lower and upper cylinders, respectively (Figure 2). The proximity sensors are magnetically mounted to the telescope enclosure's inner structure and pointed outward, such that they will be triggered as the cylinders travel up or down. (Figures 3 & 4).

2.2. Connections & Power

The motors are connected to the ClearCore motion controller at M0 (MCVC motor) and M3 (SDSK motor). The buttons and sensors have two states (on/off), are connected to the ClearCore's I/O points (see Table 1), and configured as digital inputs. The proximity sensors function as limit switches to define the bounds of each motor's motion, while the buttons allow an on-site operator to open and close the cylinders.

[2]See ClearCore user manual for the recommended power supply.

[3]McMaster-Carr Metallic-Object Proximity Switch DC, Stainless Steel, Flush with 3-Pole Plug, NPN, SKU: 73635K78. Product information

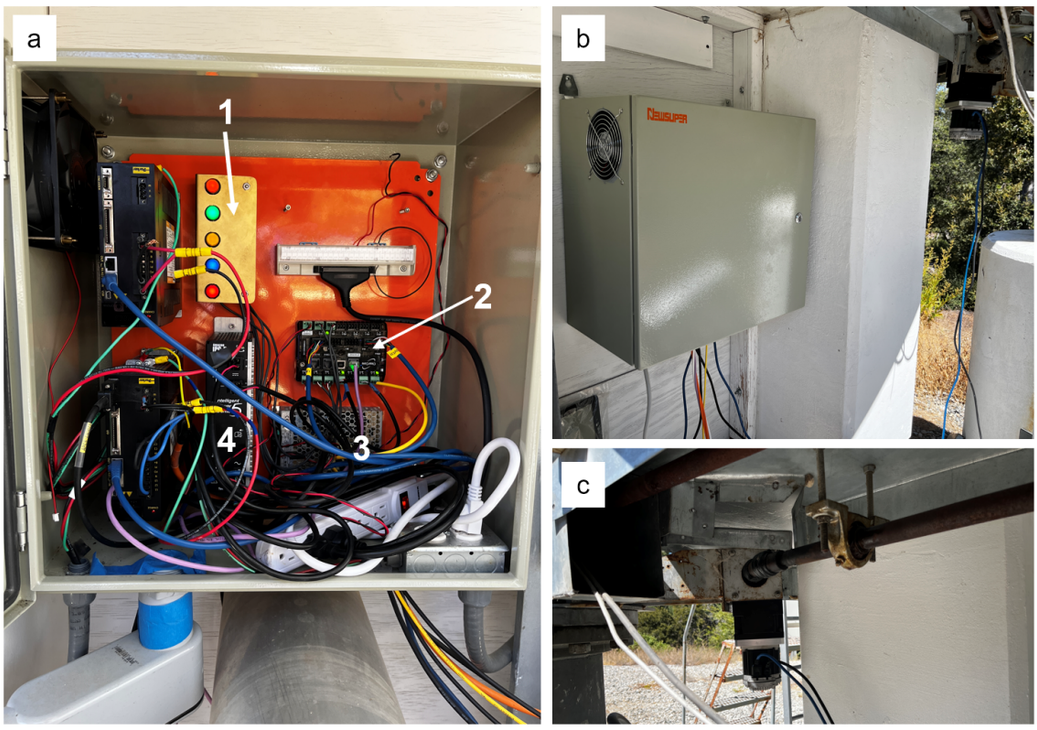

FIGURE 2. Electronics enclosure box and motors, as described in Figure 1. **(a)** Electronics board with (1) control pad, (2) ClearCore, (3) 24V power supply, (4) 75V power supply. **(b)** closed enclosure box (left) and lower motor (right). **(c)** upper motor.

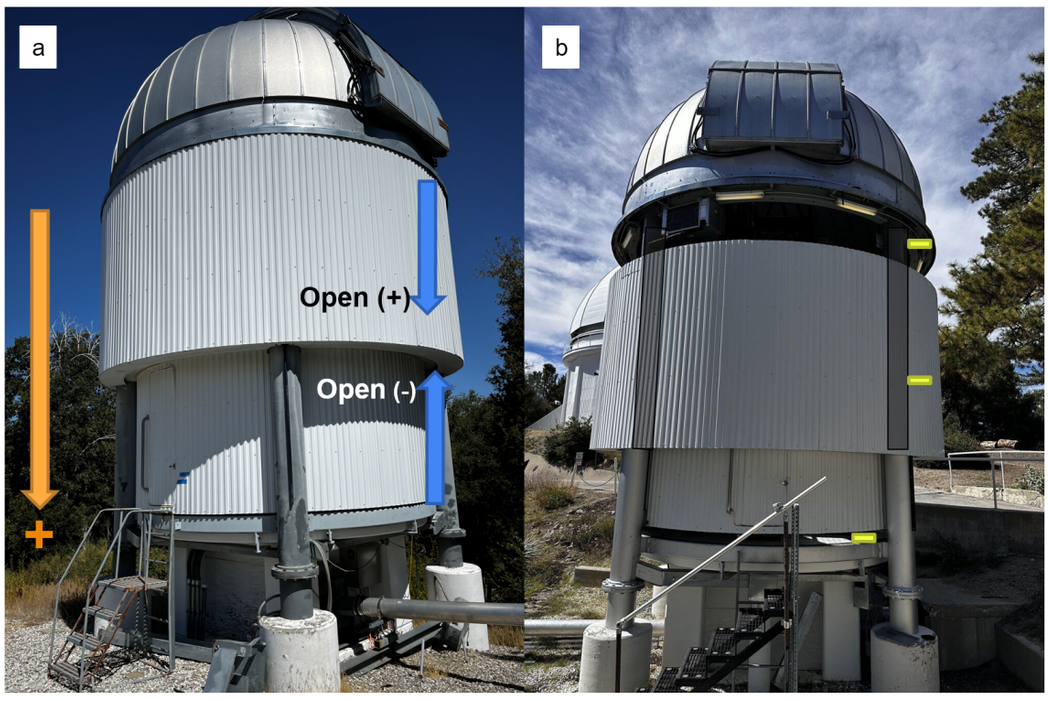

FIGURE 3. Exterior view of the telescope enclosure cylinders. **(a)** Cylinders at W1 in their closed positions. In the control software, the downwards direction is considered positive, so the upper cylinder opens in the positive direction, while the lower cylinder opens in the negative direction. **(b)** Cylinders at S2 in their open positions, with the locations of the three proximity switches marked in yellow. The semitransparent gray boxes mark the continuation of the support pillars, to which the upper home and limit switches can be mounted.

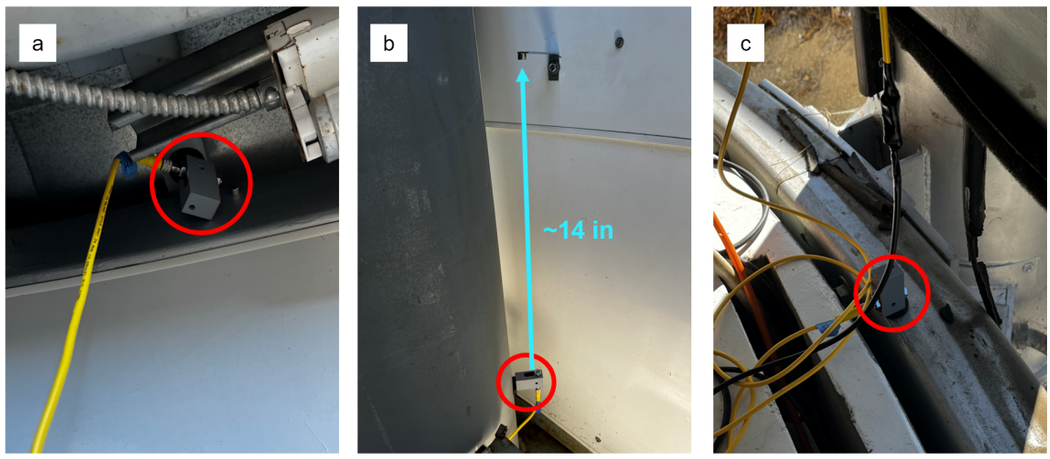

FIGURE 4. Proximity switch mounting positions. **(a)** upper home switch: underside of dome support ring. **(b)** upper limit switch: pillar, triggered by a small magnet attached to an L-bracket mounted to the inner wall of the upper cylinder. **(c)** lower home switch.

The motors are powered by the 75V IPC-5, while the ClearCore is powered by the 25V power supply. Both power supplies must be connected to an external power source. Finally, a CAT5 Ethernet cable is required to connect the ClearCore to the CHARA network.

Switch	Trigger Type	Connector	Color
Upper limit	Proximity	A-9	N/A
Upper home	Proximity	A-11	N/A
Lower home	Proximity	A-12	N/A
Upper cyl. up	On Push	DI-6	Orange
Upper cyl. down	On Push	DI-7	Green
Lower cyl. up/down	Toggle	DI-8	Yellow
Clear alerts	On Push	IO-5	Blue
Enable motors	Toggle	IO-4	Red

TABLE 1. Switches and their corresponding connector points on the ClearCore.

2.3. Moving the Cylinders

This section explains how the ClearCore and ClearPath motors control the movement of the cylinders. Instructions on operating the cylinder controls are detailed in Section 3.

2.3.1. A note on positioning

The ClearPath servo motors feature built-in optical encoders with a resolution of 800 counts per revolution. The motors use these encoders in a closed-loop system to ensure precise execution of movement commands. The encoder keeps track of position by counting the number of revolutions in either direction away from a "home" position. When the motor is "in home", the counter is at zero. As a matter of convention, we have defined opening the upper cylinder (moving it down) as a move in the positive direction, while opening the lower cylinder (moving it up) as a move in the negative direction (see Figure 3(a)). This can be conceptualized with two vertical axes, centered at the cylinders' closed positions, with the positive direction pointing down. Therefore, downwards motion is always positive for both cylinders, while upwards motion is always negative.

2.3.2. Lower Cylinder Control

The lower cylinder's MCVC motor is configured in the *Move to Absolute Position: 2 Positions (Home to Switch)* operational mode. The motor moves between two predefined positions, which translates to fixed open and closed positions for the lower cylinder. All absolute positioning commands are made in reference to the home position. The MCVC motor seeks home by slowly moving the cylinder in the closing position (downwards) until a proximity sensor placed at the bottom of the enclosure is triggered. The motor then stops, and moves a short distance in the opposite direction. Once this process is complete, the motor's internal position counter is set to zero. Thus, the cylinder's "in home" position is equivalent to its "closed" position. More information about ClearPath operational modes can be found in the ClearPath user manual.

2.3.3. Upper Cylinder Control

The upper cylinder's SDSK motor is configured in the *Step and Direction* operational mode. The motor follows step and direction commands, allowing the user to precisely and repeatably position the upper cylinder. Limits on the upper cylinder's range of motion are defined by two proximity sensors placed along one of the inner support columns. Triggering either of these sensors will halt the motor, preventing the cylinder from moving past the sensor. Unlike the MCVC motor, the SDSK motor ignores the limit sensors when homing and instead defines its home position with a *hard stop*. The motor will move the cylinder in the closing direction (upwards) until it hits the bottom edge of the dome. The motor will continue driving until it reaches its torque limit (chosen so as not to damage the mechanics), at which point it will define its current position as "home".

2.3.4. ClearCore Programming

UDP commands, as well as digital signals from the various sensors, buttons, and motors are interpreted by the ClearCore using custom programming written with Teknic's ClearCore C++ library. The program was written and uploaded to the ClearCore using Teknic's wrapper for the Arduino IDE. See Appendix B for source code and other information.

3. OPERATION

ClearCore	IP Address
S1	TBD
S2	192.168.3.221
W1	192.168.2.29
W2	TBD
E1	TBD
E2	TBD

TABLE 2. IP addresses for the ClearCores installed at each telescope as of this writing.

Each box contains a panel with buttons that can be used to control the cylinders on-site. Refer to Table 1 for the function of each button. Each ClearCore is also connected to the Internet and has a unique IP address noted in Table 2. They are programmed with custom software to receive commands from and send status reports to the operator. This allows precise and repeatable adjustments to be made in response to the weather. See Appendix A for detailed documentation on the software and the commands available to the operator.

4. DATA COLLECTION

The new system was installed at S2 and W1 because they were most affected by heating. These are each part of a pair of telescopes located at the ends of the three CHARA Array arms. Since temperatures and atmospheric conditions vary across the mountaintop, we used their closest unmodified counterparts (S1, W2) to compare cooling and DM performance.

The main metrics for cooling performance are the internal temperature of the enclosure and seeing as measured by the Fried parameter r_0. This parameter corresponds to the largest packet of stable air and has units of length (cm). Temperature sensors ("Minions") were placed at the deformable mirror (DM), the M5 mirror, and next to the Telescope Management (TEMA) (Figure 5). This allows us to measure the vertical temperature gradient across the adaptive optics board (AOB) and the telescope enclosure as a whole. Seeing (r_0) is measured by the Shack-Hartmann wavefront sensor (WFS) located on the AOB between the DM (M4) and M5.

Installation of the new system at S2 was completed and the cylinders became operational on July 11th, 2024, and data collection began on the same date. The cylinders at W1 became operational on July 18th, 2024. Data collection began on July 19th. The Minions provided around-the-clock temperature monitoring,

5. PRELIMINARY RESULTS & DISCUSSION

The preliminary trial involved opening only the lower cylinders at S2 and W1 during observing every night from July 31st to August 6th, except for August 1st (we will call this the trial period). We calculated the twenty-minute rolling averages of r_0 for each telescope and

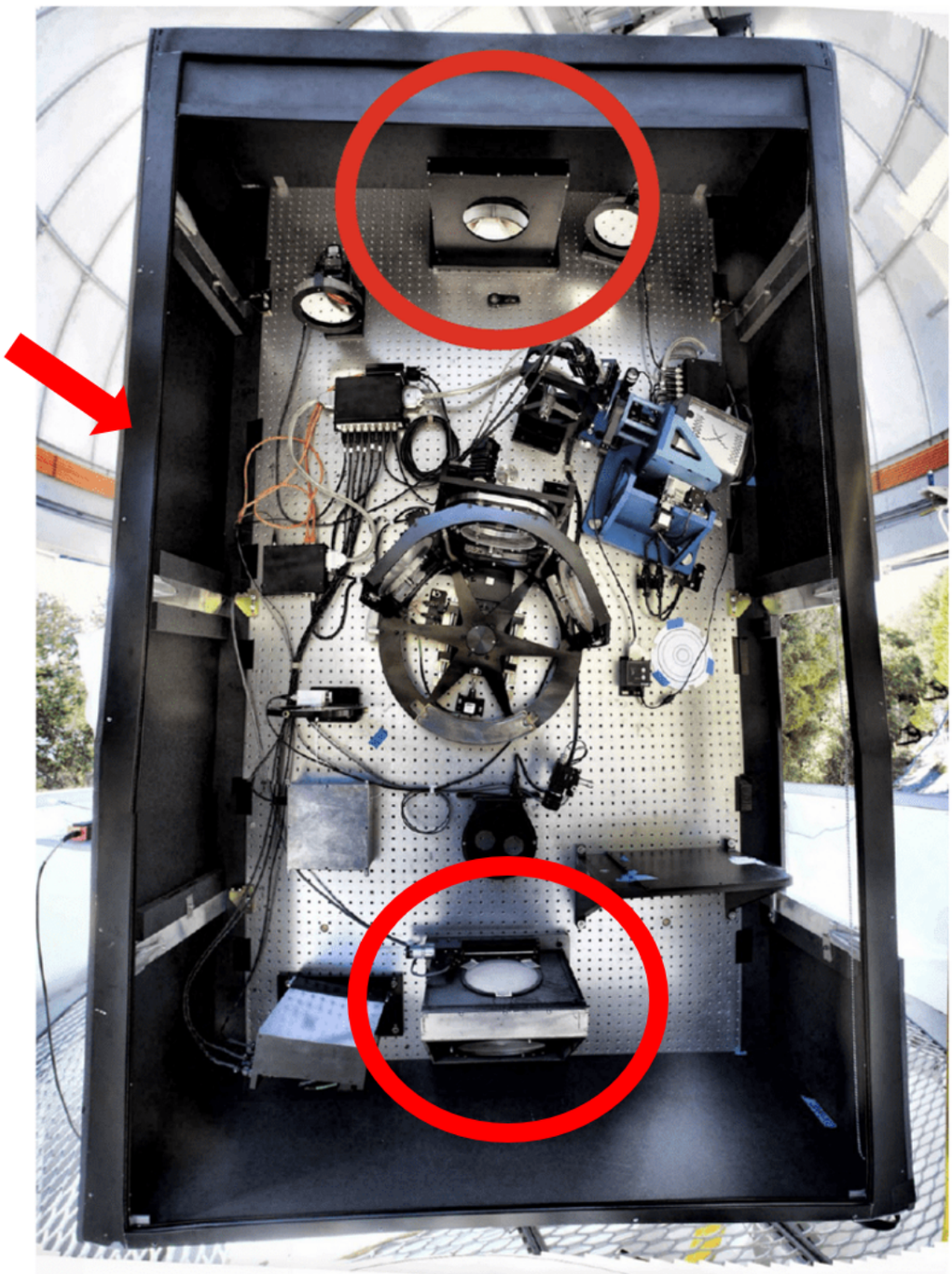

FIGURE 5. Adaptive Optics Board (AOB) with temperature sensor locations marked in red. Top circle: DM; bottom circle: M5; Arrow: TEMA (outside of the AOB). Note that temperature sensors are not present in this picture. The black box surrounding the AOB is closed during observing.

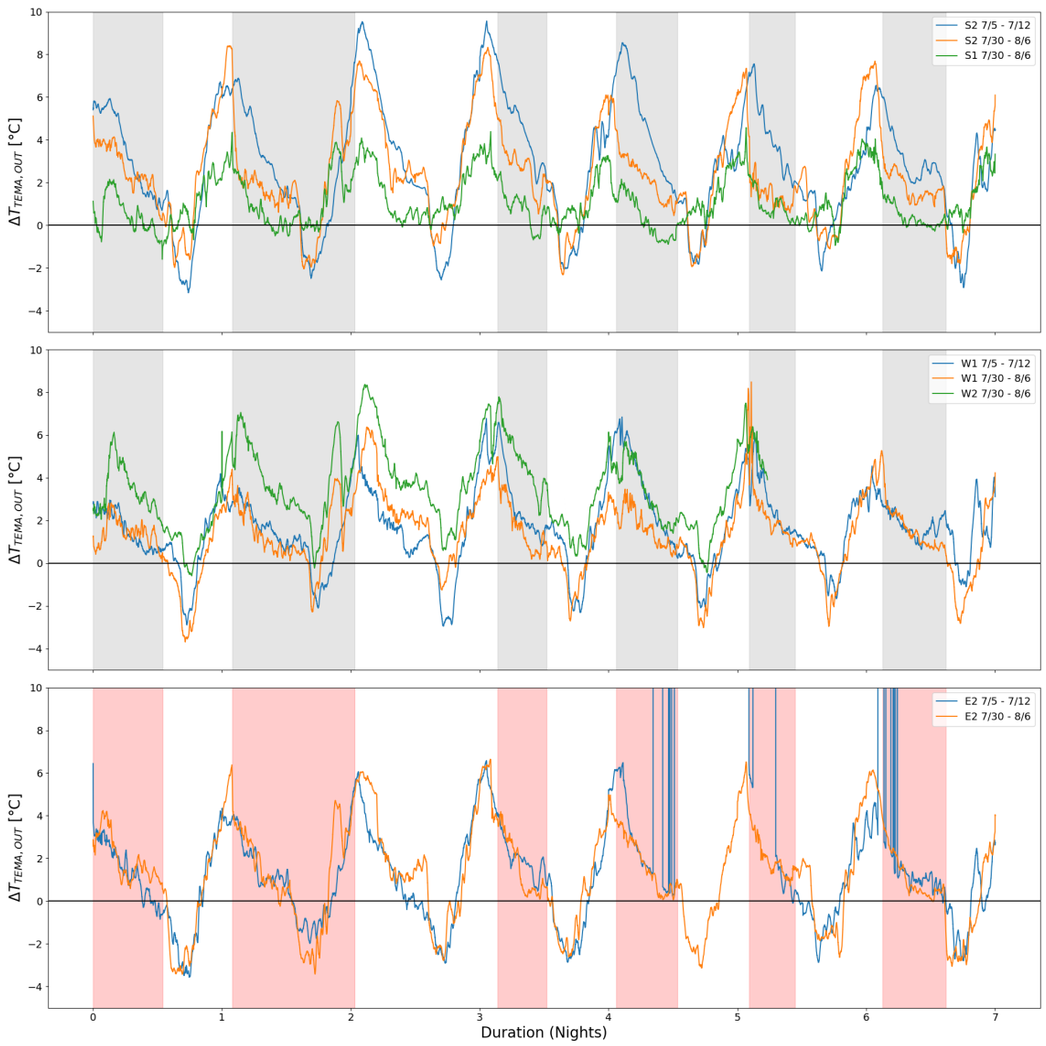

FIGURE 6. $\Delta T_{TEMA,OUT}$ for all three telescope pairs (S1-S2, W1-W2, and E1-E2) during the control and trial periods. The grey shaded regions mark nights during the trial period where the lower cylinders were opened at S2 and W1. The red shaded regions mark the same nights at E2/E1, but the cylinders were not actually opened. **Top:** $\Delta T_{TEMA,OUT}$ at S2 during the trial period (orange) compared with itself during the control period (blue) and S1 during the trial period (green). **Middle:** Same as top panel, but comparing W1 with its past cooling performance and that of W2. **Bottom:** Same as top and middle panels. No data for $\Delta T_{TEMA,OUT}$ at E1 could be included due to a missing sensor.

the differences between their exterior and interior (DM/TEMA) temperatures ($\Delta T_{DM,OUT}$ and $\Delta T_{TEMA,OUT}$) during this period. We focus on the temperature difference rather than the absolute temperatures due to variations in exterior temperatures at each telescope, and we seek to minimize these values. We compared these to the seven-night period from July 5th to July 12th, before the cylinders were operational (we will call this the control period). We also compared these parameters for each telescope pair during the same time periods.

5.1. Effects on Cooling

The telescope enclosures are sealed during daytime, leading to heating of the inside air. Prior to observing, the domes at each telescope are opened to vent the hot air. With only the lower cylinders open, S2 and W1 both showed signs of improved cooling compared to E2. When the cylinders were open, $\Delta T_{TEMA,OUT}$ was rarely greater than ≈ 4°C at S2 and W1 (6) At S2, the cooling effect is especially noticeable. During the trial period, nights often begin at a $\Delta T_{TEMA,OUT}$ seen only toward the end of nights during the control period. S2 is generally hotter than S1; nevertheless, opening the cylinders allows S2 to mitigate its significant daytime heating and approach S1's nighttime temperature to within ≈ 1°C. At W1, the effect is less pronounced, but opening the cylinders still enabled W1 to reach lower temperatures sooner than it had in the control period. This is also an improvement compared to W2, which is generally hotter than W1. We compare these results to $\Delta T_{TEMA,OUT}$ at E2, whose cylinders were not open during the control or trial

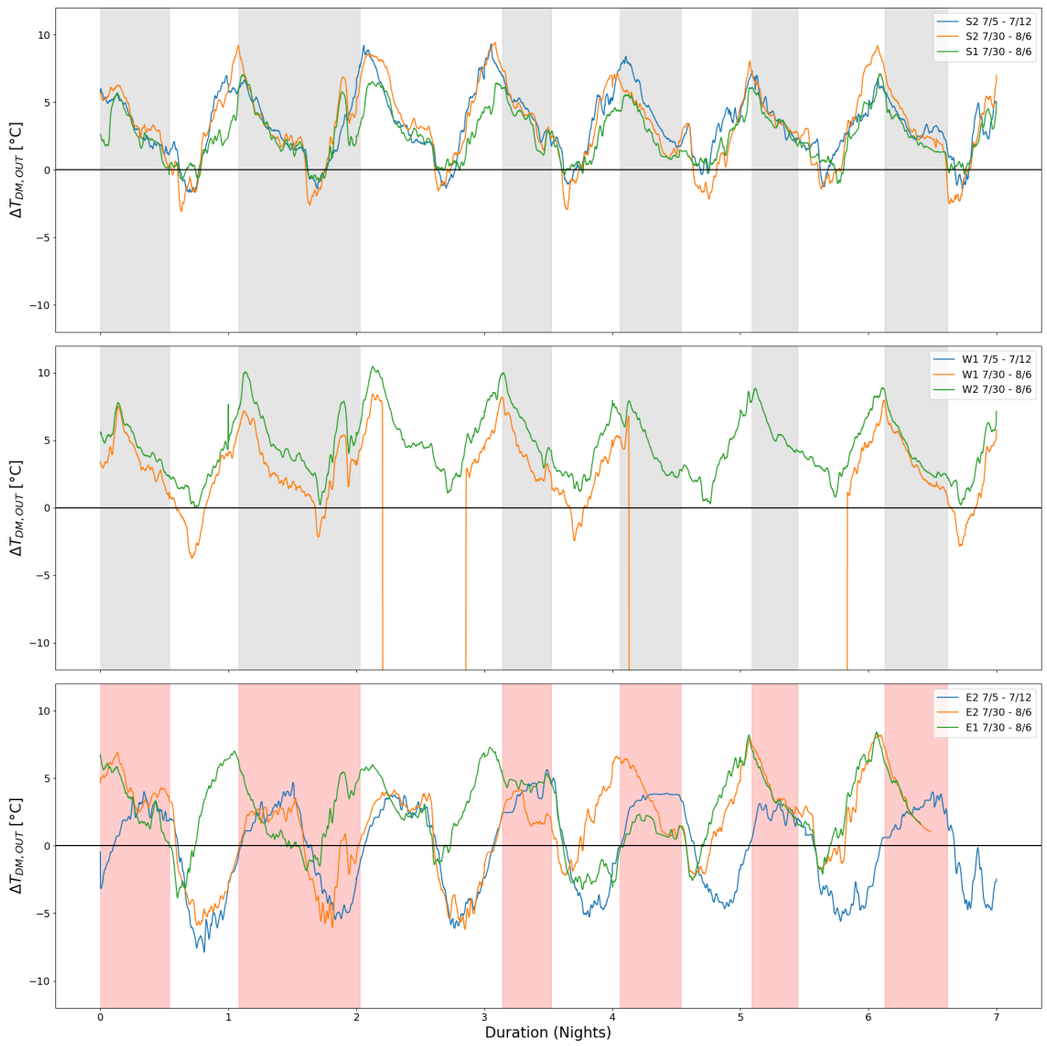

FIGURE 7. Data for $\Delta T_{DM,OUT}$ presented in the same manner as Figure 6.

periods. As seen in Figure 6, there is no significant difference in cooling rate between the two periods.

Cooling effects at the DM are not as dramatic. This is due in part to its location inside the AOB, which is closed off from external light interference by a black box (see Figure 5 and is thus somewhat insulated from the rest of the enclosure's interior. However, Figure 7 shows that at S2, opening the lower cylinder brings its $\Delta T_{DM,OUT}$ closer to that of S1. More testing data, consisting of a longer control and trial period, is needed to determine the cooling effects at the DM, as well as at M5.

5.2. Effects on Seeing

Telescope	Control r_0 [cm]	Trial r_0 [cm]	Difference [cm]	Difference[%]
S1	13.539	14.539	1.000	7.387
S2	17.870	18.477	0.607	3.397
W1	12.008	13.041	1.033	8.597
W2	14.854	15.802	0.948	6.384
E1	18.268	18.600	0.332	1.815
E2	14.772	15.819	1.047	7.088

TABLE 3. 7-night average r_0 during the trial and control periods, their differences in cm, and their percent differences.

Although r_0 is roughly anti-correlated with air temperature (Figure 8), there are many other factors, such as relative humidity and wind speed, that also contribute to the stability of the air between a given telescope and its target. Preliminary data does not suggest a significant cooling effect on the DM for both S2 and W1, so we did not expect to see a significant difference in seeing (r_0). Here, we compare the trial period (7/31 - 8/6) with

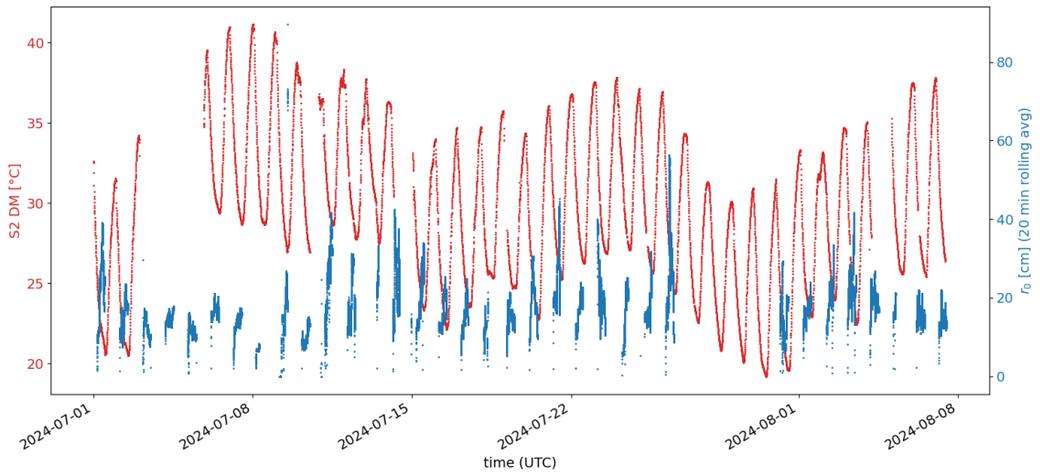

FIGURE 8. Daily variation in r_0 and DM temperature at S2 from July to August 2024.

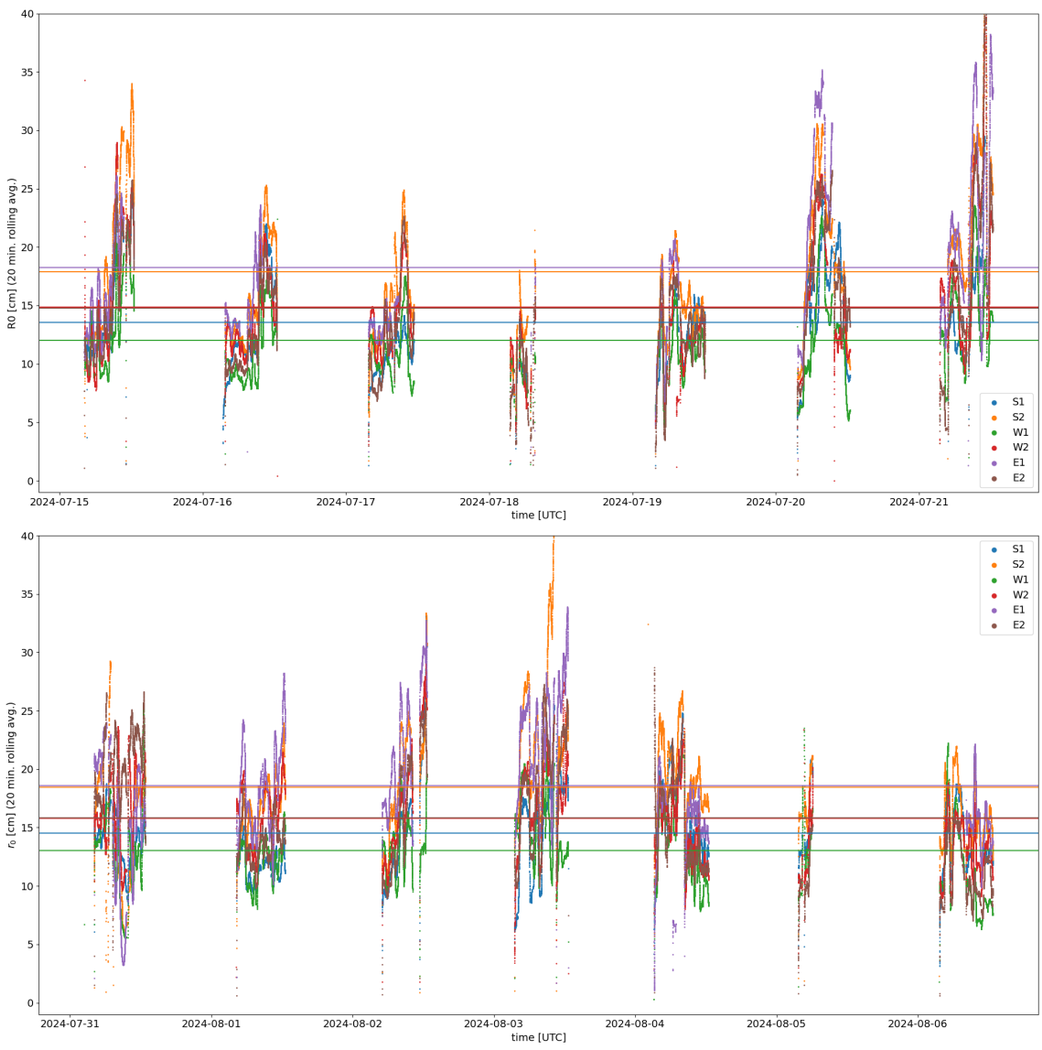

FIGURE 9. 20-minute rolling average of r_0 derived from each telescope's WFS over two seven-night periods. The horizontal lines indicate the average r_0 for each telescope during that period.

a later 7-night control period (7/15 - 7/21) than in Section 5.1. The control period was chosen to have approximately the same average exterior temperature as the trial period (a difference of ≈ 0.1°C across all telescopes) so that the dependence of r_0 on the interior temperature could be seen more clearly. Figure 9 shows the change in the 20-minute rolling average of r_0 throughout each night as well as the 7-night averages for each telescope. The rolling window was chosen to reduce noise in the WFS data. We see an 8.6% increase in the 7-night average r_0 for W1 and a 3.4% increase for S2. However, telescopes with non-functioning cylinders also showed increases of the same magnitude (Table 3). Further testing is required; opening the upper cylinder may result in a greater increase in average r_0 by allowing hot air to vent before rising into the optical path of the telescope.

6. SUMMARY

In this Technical Report, we have described the design and construction of a new cylinder motion control system to be implemented at all six CHARA Array telescopes. This system enables telescope operators to open additional vents on the sides of the enclosures by moving the cylinders, which leads to increased passive cooling. The new system is installed and operational at S2 and W1. The next system will be installed at E1 or E2. We expect a well-cooled telescope to reach its equilibrium temperature more rapidly, leading to an increase in r_0 at the beginning of a given night, but were unable to demonstrate this yet. Having the ability to open and close the cylinders at all six telescopes will allow us to conduct more thorough tests of this hypothesis. Although we have yet to measure a conclusive increase in r_0, being able to cool the dome more effectively means that mirror alignment, which is sensitive to large temperature swings, will be more stable and require less correction throughout the night. Additionally, opening the upper cylinder allows access to the main telescope mirror for re-coating. Finally, a cooler dome will result in greater stability of the optical system, less wear on components and electronics, and a more comfortable daytime work environment.

A. DETAILS ON REMOTE OPERATION

Cylinders controlled by a ClearCore (currently S2 and W1) can be remotely operated by connecting to their IP address on a computer connected to the CHARA network.

A.1. Connecting

In the terminal, connect to the ClearCore through UDP using netcat: `nc -u <ipaddress> 8888`. Replace `<ipaddress>` with the actual IP of the ClearCore according to Table 2.

A.2. Controlling

Send commands to the ClearCore through the UDP connection. The following commands are available:

```
m#: move upper # steps (+/-), 800 steps/revolution
U: Lower Open, D: Lower Close,
O: Upper Open, C: Upper Close,
a: abort move, x: clear alerts,
s#: set #steps (+/-), v#: set velocity (+/-, limit 5333),
S: Status, M: upper cyl. movement settings
e/d: enable/disable motors
```

To send a command, type it in the command line and press enter. You will receive the above text as a reply packet every time you send a command. Please note the following:

- Position/velocity is measured in motor encoder counts/steps.
  - The upper cylinder can be positioned anywhere between 0 and 4,000,000 steps (about 12 inches of linear motion). If a move command would take the upper motor out of this range, it will move to the min/max allowed position.
  - Lower cylinder: per-step control not available. Currently moves about 12 inches (2,000,000 steps) between "open" and "closed" positions.
- The direction of positive motion is downward. Physically, the upper cylinder moves DOWN (+) to open, and the lower cylinder moves UP (-) to open.
  - e.g. 'm1000000' opens the upper cylinder by moving it downward by 1,000,000 steps. Opening the lower cylinder moves it upward by -2,000,000 steps.
  - Both cylinders have 'position = 0' when closed.
- While the cylinders are moving, the ClearCore will send `Waiting for HLFB...` through the UDP connection. When they have reached their commanded position, ClearCore will send `Move Done`. While the cylinders are moving, the ClearCore will only execute the following commands:
  - `d` (disable), `a` (abort), and `S` (status).
  - **Known issue:** when opening the lower cylinder, ClearCore will report `Move Done` without waiting for it to finish moving. Closing does not have this issue. Confirm by checking the status report.

A.3. Checking Status

Send `S` to get the status report. It should look like this:

```
                  S2 CYLINDER STATUS
=================================================
 Upper cyl. position:      MaxOpen (4000000 steps)
 Lower cyl. position:      Closing

Button              State | Sensor             State
------------------------+-------------------------
 Upper ctrl. up      OFF | Upper cyl. home      OFF
 Upper ctrl. down    OFF | Upper cyl. limit     ON
 Lower ctrl.         OFF |
 Clear alerts        OFF | Lower cyl. home      OFF
 Enable motors       ON  | Lower cyl. limit     OFF

Alerts
-------------------------------------------------
Upper motor:
Lower motor:
```

The first two rows report the cylinder positions. "Open" and "Closed" mean that the cylinders are open/closed and stationary. "Opening" and "Closing" mean that the cylinders are in motion toward the open/closed position. "MaxOpen" is exclusive to the upper cylinder and means it cannot open any further.

The status report also contains the momentary state of each button/sensor. The buttons are hardware controls located at the telescope. The sensors are proximity switches that are ON by default and turn OFF when triggered. Homing and limit sensors are placed at the cylinders' closed and open positions respectively. If a given sensor is OFF, the cylinder cannot be moved further in that direction.

Known Issue: the sensors may not be triggered even when the cylinders are fully closed/open.

Note: "Lower cyl. limit" sensor is currently not in use; please ignore it.

If there are any alerts, clear them by sending `x`.

B. CLEARCORE CONTROL SOURCE CODE

The following C code is located in `UDP_bothcylinders_ClearCore8.ino` hosted in the cylinders repository on the CHARA GitLab.

Editing and compiling the code requires Teknic's ClearCore wrapper for the Arduino IDE. Download and installation instructions can be found on the Teknic ClearCore API documentation page. This page also contains documentation for each class in the ClearCore API. Uploading the code to the ClearCore requires the Arduino IDE and a USB-A to USB-B cable.

```
/* Control of Upper and Lower Cylinders for CHARA telescope enclosures.
  Authors: James Wen (jswen@usc.edu), Nic Scott (nscott14@gsu.edu)

   1. Teknic ClearPath motors must be connected to the ClearCore
      (MC to Connector M-0, SD to Connector M-3). MC controls the
      lower cylinder, and SD controls the upper cylinder.

   2. The connected ClearPath MC motor must be configured
      through the MSP software for Move To Absolute Position mode
      (In MSP select Mode>>Position>>Move to Absolute Position).

      The connected ClearPath SD motor must be configured
      through the MSP software for Step and Direction mode
      (In MSP select Mode>>Step and Direction).

   3. Homing must be configured in the MSP software for your mechanical system
      (e.g. homing direction, switch polarity, etc.). To configure, click
      "Setup..." found under the "Homing" label on the MSP's main window.

   4. The ClearPath motor must be set to use the HLFB mode "ASG-Position
      w/Measured Torque" with a PWM carrier frequency of 482 Hz through the MSP
      software (select Advanced>>High Level Feedback [Mode]... then choose
      "ASG-Position w/Measured Torque" from the dropdown, make sure that 482 Hz
      is selected in the "PWM Carrier Frequency" dropdown, and hit the OK
      button). Set the Input Format in MSP for "Step + Direction" (for the SD motor).

```

```
    5. Wire the homing sensor to Connector (for the MC motor).

    6. The ClearPath must have defined Absolute Position Selections set up in the
       MSP software (On the main MSP window check the "Position Selection Setup
       (cnts)" box and fill in the two text boxes labeled "A off" and "A on").

    7. Ensure the Input A & B filters in MSP are both set to 20ms (In MSP
       select Advanced>>Input A, B Filtering... then in the Settings box fill in
       the textboxes labeled "Input A Filter Time Constant (msec)" and "Input B
       Filter Time Constant (msec)" then hit the OK button).

    Links:
 * ClearCore Documentation: https://teknic-inc.github.io/ClearCore-library/
 * ClearCore Manual: https://www.teknic.com/files/downloads/clearcore_user_manual.pdf
 * ClearPath Manual: https://www.teknic.com/files/downloads/clearpath_user_manual.pdf
 * ClearPath Mode Informational Video: https://www.teknic.com/watch-video/#OpMode2
*/

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "ClearCore.h"
#include "EthernetUdp.h"

// Change the MAC address and IP address below to match your ClearCore's MAC address and IP.
// byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
// byte mac[] = {0x24, 0x15, 0x10, 0xB0, 0x2A, 0xF4};
IpAddress ip = IpAddress(192, 168, 2, 134);  //set if configured for static IP
unsigned int localPort = 8888; // The local port to listen for connections on.

// buffer to read incoming UDP packets
#define MAX_PACKET_LENGTH 10
unsigned char packetReceived[MAX_PACKET_LENGTH];
// buffer to write outgoing UDP packets
char inttostr[10];
// An EthernetUDP instance to let us send and receive packets over UDP
EthernetUdp Udp;
// Set this false if not using DHCP to configure the local IP address.
bool usingDhcp = true;
bool enabled = false;

// Specify which serial to use (ConnectorUsb, ConnectorCOM0, or ConnectorCOM1) and the baud rate.
#define SerialPort ConnectorUsb
#define baudRate 9600

// The INPUT_A_B_FILTER must match the Input A, B filter setting in MSP
// (Advanced >> Input A, B Filtering...)
#define INPUT_A_B_FILTER 20

// Defines the motor's connectors
#define lower_motor ConnectorM0  // MC motor
#define upper_motor ConnectorM3  // SD motor
// Specifies the home sensor connector
#define lower_HomingSensor ConnectorA12
#define upper_HomingSensor ConnectorA11
// Specifies the Limit switch connectors
#define lower_LimitSwitch ConnectorA10
#define upper_LimitSwitch ConnectorA9
// Specifies buttons
#define lower_ctrl_button ConnectorDI8       // lower cylinder cycle Aoff/Aon (latching)
#define upper_ctrl_up_button ConnectorDI6    // upper cylinder dir1 (momentary)
#define upper_ctrl_down_button ConnectorDI7  // upper cylinger dir2 (momentary)
#define enable_button ConnectorIO4           // enable/disable (latching)
#define clear_button ConnectorIO5            // clear alerts (momentary)

// current state of button pins
int16_t lower_ctrl_state;
int16_t upper_ctrl_up_state;
int16_t upper_ctrl_down_state;
int16_t enable_state;
int16_t clear_state;
int16_t last_lower_ctrl_state = false;
int16_t last_upper_ctrl_up_state = false;
int16_t last_upper_ctrl_down_state = false;
int16_t last_enable_state = false;
int16_t last_clear_state = false;

// limit/home switch interrupt routines
void lower_HomingSensor_Callback();
void upper_HomingSensor_Callback();
void lower_LimitSwitch_Callback();  //lower cylinder upper limit switch
void upper_LimitSwitch_Callback();  //upper cylinder upper limit switch

```

```
// movement and communication routines
bool UpperMoveDistance(int32_t distance);
bool LowerMovePosition(uint8_t positionNum);
bool UpperMoveAtVelocity(int32_t velocity);
void PrintAlerts();
void HandleAlerts();
void printStatus();
void printUpperMoveSettings();
void SerialUdpSendln(char *message);
uint16_t packetSize;

// motor/sensor states for status report
char *lowerPosition;
char *upperPosition;
char statusReport[1024];
char upperCtrlUpState_str[4] = "OFF";
char upperCtrlDownState_str[4] = "OFF";
char lowerCtrlState_str[4] = "OFF";
char clearState_str[4] = "OFF";
char enableState_str[4] = "OFF";
char upperHomeState_str[4] = "OFF";
char upperLimitState_str[4] = "OFF";
char lowerHomeState_str[4] = "OFF";
char lowerLimitState_str[4] = "OFF";

// Define the position, velocity, and acceleration limits to be used for each move
int cylinder_range = 4000000; //change this to match actual number of steps from home to limit
int32_t velocityLimit = 5333; // pulses per sec 800 pulses/rev, 400 rpm = 5333 pulses/sec
int32_t accelerationLimit = 1333; // pulses per sec^2, 100 rpm/s
// initial movement settings
int vel = velocityLimit;
int steps = 1000;
int distance = 1000;
int upper_homing_offset = 0; // should match "Homing Offset Move Distance (cnts)" in the MSP homing setup menu
bool stop = true;
bool lower_pos;
bool abort_move = false;
bool move = false;
int travel = 0;
int i = 0;


// To enable automatic alert handling, #define HANDLE_ALERTS (1)
// To disable automatic alert handling, #define HANDLE_ALERTS (0)
#define HANDLE_ALERTS (1)

// Put your setup code here, it will run once:
void setup() {
  uint32_t timeout = 5000;
  uint32_t startTime = Milliseconds();

  // Sets up serial communication and waits up to 5 seconds for a port to open.
  SerialPort.Mode(Connector::USB_CDC);
  SerialPort.Speed(baudRate);
  SerialPort.PortOpen();
  while (!SerialPort && Milliseconds() - startTime < timeout) {
    continue;
  }

  // Make sure the physical link is up before continuing.
  while (!EthernetMgr.PhyLinkActive() && Milliseconds() - startTime < timeout) {
    Serial.println("The Ethernet cable is unplugged...");
    Delay_ms(1000);
  }
  // Run the setup for the ClearCore Ethernet manager.
  EthernetMgr.Setup();
  if (usingDhcp) {
    // Use DHCP to configure the local IP address.
    bool dhcpSuccess = EthernetMgr.DhcpBegin();
    if (dhcpSuccess) {
      Serial.println("DHCP successfully assigned an IP address: ");
      Serial.println(EthernetMgr.LocalIp().StringValue());
    } else {
      Serial.println("DHCP configuration was unsuccessful!");
      while (true && Milliseconds() - startTime < timeout) {
        // UDP will not work without a configured IP address.
        continue;
      }
    }
  } else {
    EthernetMgr.LocalIp(ip);
    Serial.println("Successfully assigned a static IP address: ");
    Serial.println(EthernetMgr.LocalIp().StringValue());
```

```
  }
  // Begin listening on the local port for UDP datagrams
  Udp.Begin(localPort);

  // This section attaches the interrupt callback to the homing/limit sensor pins,
  // set to trigger when the sensor goes from on to off.

  lower_HomingSensor.Mode(Connector::INPUT_DIGITAL);
  lower_HomingSensor.InterruptHandlerSet(lower_HomingSensor_Callback, InputManager::CHANGE);
  upper_HomingSensor.Mode(Connector::INPUT_DIGITAL);
  upper_HomingSensor.InterruptHandlerSet(upper_HomingSensor_Callback, InputManager::FALLING);
  lower_LimitSwitch.Mode(Connector::INPUT_DIGITAL);
  lower_LimitSwitch.InterruptHandlerSet(lower_LimitSwitch_Callback, InputManager::FALLING);
  upper_LimitSwitch.Mode(Connector::INPUT_DIGITAL);
  upper_LimitSwitch.InterruptHandlerSet(upper_LimitSwitch_Callback, InputManager::FALLING);
  InputMgr.InterruptsEnabled(true);

  // Sets the input clocking rate. This normal rate is ideal for ClearPath
  // step and direction applications.
  MotorMgr.MotorInputClocking(MotorManager::CLOCK_RATE_NORMAL);
  // Sets all motor connectors to the correct modes
  MotorMgr.MotorModeSet(MotorManager::MOTOR_M0M1, Connector::CPM_MODE_A_DIRECT_B_DIRECT);
  MotorMgr.MotorModeSet(MotorManager::MOTOR_M2M3, Connector::CPM_MODE_STEP_AND_DIR);
  // Set the motor's HLFB mode to bipolar pwm
  lower_motor.HlfbMode(MotorDriver::HLFB_MODE_HAS_BIPOLAR_PWM);
  upper_motor.HlfbMode(MotorDriver::HLFB_MODE_HAS_BIPOLAR_PWM);
  // Set the HFLB carrier frequency to 482 Hz
  lower_motor.HlfbCarrier(MotorDriver::HLFB_CARRIER_482_HZ);
  upper_motor.HlfbCarrier(MotorDriver::HLFB_CARRIER_482_HZ);

  // Enforces the state of the motor's Input A/B before enabling the motor
  lower_motor.MotorInAState(false);
  lower_motor.MotorInBState(lower_HomingSensor.State());
  // upper_motor.MotorInBState(true);

  // Set velocity and acceleration limits
  upper_motor.VelMax(velocityLimit);
  upper_motor.AccelMax(accelerationLimit);

  // Set pos and neg limit switches for upper motor (S&D only)
  // NOTE: cannot use declared macros for clearcore pins.
  if (upper_motor.LimitSwitchPos(CLEARCORE_PIN_A9)) {
    Serial.println("Upper home sensor enabled on pin A11");
  }
  if (upper_motor.LimitSwitchNeg(CLEARCORE_PIN_A11)) {
    Serial.println("Upper limit switch enabled on pin A9");
  }

  printStatus();
}

void loop() {
  // Put your main code here, it will run repeatedly:

  // Look for a received packet.
  packetSize = Udp.PacketParse();

  if (packetSize > 0) {
    SerialPort.Send("Received packet of size ");
    SerialPort.Send(packetSize);
    SerialPort.Send(" bytes. ");
    SerialPort.Send("Remote IP: ");
    SerialPort.SendLine(Udp.RemoteIp().StringValue());
    SerialPort.Send("Remote port: ");
    SerialPort.SendLine(Udp.RemotePort());
    // Read the packet.
    int32_t bytesRead = Udp.PacketRead(packetReceived, MAX_PACKET_LENGTH);
    SerialPort.Send("Number of bytes read from packet: ");
    SerialPort.SendLine(bytesRead);
    SerialPort.Send("Packet contents: ");
    SerialPort.Send((char *)packetReceived);
    SerialPort.SendLine();
    // Send a reply packet back to the sender.
    Udp.Connect(Udp.RemoteIp(), Udp.RemotePort());
    Udp.PacketWrite(
      "\r\nm#: move upper # steps (+/-), 800 steps/revolution"
      "\r\nU: Lower Open, D: Lower Close, "
      "\r\nO: Upper Open, C: Upper Close, "
      "\r\na: abort move, x: clear alerts, "
      "\r\ns#: set #steps (+/-), v#: set velocity (+/-, limit 5333), "
      "\r\nS: Status, M: upper cyl. movement settings,"
      "\r\ne/d: enable/disable motors\r\n\r\n");
    Udp.PacketSend();
```

```
276
277       char mode = packetReceived[0];
278       char init[0];
279       int travel = 0;
280       int i = 0;
281
282       // Define commands to be read from UDP. will only run case A or B after case 'r' for reset.
283       // B will take anything (currently 9 digits) after the first char and make it an int.
284       // This could be the #steps for the motor
285       switch (mode) {
286         case 'U':
287           if (enabled == true) {
288             LowerMovePosition(2);  // Moves cylinder up to Position 2 (open) defined in MSP.
289             Delay_ms(1000);
290           } else {
291             SerialUdpSendln("Is the motor enabled?");
292           }
293           break;
294
295         case 'D':
296           if (enabled == true) {
297             LowerMovePosition(1);  // Moves cylinder down to Position 1 (closed) defined in MSP.
298             Delay_ms(1000);
299           } else {
300             SerialUdpSendln("Is the motor enabled?");
301           }
302           break;
303
304         case 'O':
305           if (enabled) {
306             UpperMoveDistance(cylinder_range);
307           } else {
308             SerialUdpSendln("Is the motor enabled?");
309           }
310           break;
311
312         case 'C':
313           if (enabled) {
314             UpperMoveDistance(-cylinder_range);
315           } else {
316             SerialUdpSendln("Is the motor enabled?");
317           }
318           break;
319
320         case 'x':
321           HandleAlerts();
322           Delay_ms(1000);
323           break;
324
325         case 'e':
326           lower_motor.EnableRequest(true);
327           upper_motor.EnableRequest(true);
328           // Waits for HLFB to assert (waits for homing to complete if applicable)
329           SerialUdpSendln("Waiting for Lower Motor...");
330           while (
331             lower_motor.HlfbState() != MotorDriver::HLFB_ASSERTED &&
332             lower_HomingSensor.State()
333           ) {
334             continue;
335           }
336           SerialUdpSendln("Lower Motor Ready");
337           SerialUdpSendln("Waiting for Upper Motor...");
338           while (upper_motor.HlfbState() != MotorDriver::HLFB_ASSERTED) {
339             continue;
340           }
341           SerialUdpSendln("Homing complete.");
342           // set the upper motor's position reference after homing is complete.
343           upper_motor.PositionRefSet(upper_homing_offset);
344           SerialUdpSendln("Upper Motor Ready");
345
346           enabled = true;
347           abort_move = false;
348           break;
349
350         case 'd':
351           SerialUdpSendln("Aborting moves ...");
352           lower_motor.MoveStopDecel(100000);
353           upper_motor.MoveStopDecel(100000);
354
355           lower_motor.EnableRequest(false);
356           upper_motor.EnableRequest(false);
357           enabled = false;
358           SerialUdpSendln("Motors disabled");
```

```
359           break;
360
361         case 'm':
362           if (enabled == true) {
363             sscanf(reinterpret_cast<const char *>(packetReceived), "%1c %9d", init, &distance);
364             upper_motor.VelMax(vel);
365             if (abs(distance) <= cylinder_range) {
366               UpperMoveDistance(distance);
367             } else {
368               SerialUdpSendln("\r\nError: requested move out of range.\r\n");
369             }
370             Delay_ms(1000);
371           } else {
372             SerialUdpSendln("Is the motor enabled?");
373           }
374           break;
375
376         case 'v':
377           sscanf(reinterpret_cast<const char *>(packetReceived), "%1c %9d", init, &vel);
378           if (abs(vel) <= velocityLimit) {
379             SerialPort.Send("Upper cyl. velocity set to ");
380             SerialPort.SendLine(vel);
381             Udp.Connect(Udp.RemoteIp(), Udp.RemotePort());
382             Udp.PacketWrite("\r\nUpper cyl. velocity set to ");
383             Udp.PacketWrite(itoa(vel, inttostr, 10));
384             Udp.PacketWrite("\r\n");
385             Udp.PacketSend();
386           } else {
387             SerialUdpSendln("\r\nError: requested velocity over limit.\r\n");
388           }
389           Delay_ms(1000);
390           break;
391
392         case 'a':
393           SerialUdpSendln("Abort move");
394           abort_move = true;
395           upper_motor.MoveStopDecel(100000);
396           lower_motor.MoveStopDecel(100000);
397           Delay_ms(1000);
398           break;
399
400         case 's':
401           sscanf(reinterpret_cast<const char *>(packetReceived), "%1c %9d", init, &steps);
402           SerialPort.SendLine("steps = ");
403           SerialPort.SendLine(steps);
404           Udp.Connect(Udp.RemoteIp(), Udp.RemotePort());
405           Udp.PacketWrite("\r\nsteps = ");
406           Udp.PacketWrite(itoa(steps, inttostr, 10));
407           Udp.PacketWrite("\r\n");
408           Udp.PacketSend();
409           Delay_ms(1000);
410           break;
411
412         case 'S':
413           printStatus();
414           Delay_ms(1000);
415           break;
416
417         case 'M':
418           printUpperMoveSettings();
419           Delay_ms(1000);
420           break;
421
422         default:
423           SerialUdpSendln("No case found");
424           break;
425       }
426     }
427
428     // clear the packet buffer
429     for(int i=0;i<MAX_PACKET_LENGTH;i++) packetReceived[i] = 0;
430
431
432     lower_ctrl_state = lower_ctrl_button.State();
433     upper_ctrl_up_state = upper_ctrl_up_button.State();
434     upper_ctrl_down_state = upper_ctrl_down_button.State();
435     enable_state = enable_button.State();
436     clear_state = clear_button.State();
437
438     if (enable_state != last_enable_state) {
439       last_enable_state = enable_state;
440       if (enable_state) {
441         lower_motor.EnableRequest(true);
```

```
442         upper_motor.EnableRequest(true);
443         // Waits for HLFB to assert (waits for homing to complete if applicable)
444         SerialUdpSendln("Waiting for Lower Motor...");
445         while (
446           lower_motor.HlfbState() != MotorDriver::HLFB_ASSERTED &&
447           lower_HomingSensor.State()
448         ) {
449           continue;
450         }
451         SerialUdpSendln("Lower Motor Ready");
452         // SerialUdpSendln("Waiting for Upper Motor...");
453         // while (upper_motor.HlfbState() != MotorDriver::HLFB_ASSERTED) {
454         //   continue;
455         // }
456         SerialUdpSendln("Homing complete.");
457         // set the upper motor's position reference after homing is complete.
458         upper_motor.PositionRefSet(upper_homing_offset);
459         SerialUdpSendln("Upper Motor Ready");
460         enabled = true;
461         SerialUdpSendln("Motors enabled");
462       } else {
463         SerialUdpSendln("Aborting moves ...");
464         lower_motor.MoveStopDecel(100000);
465         upper_motor.MoveStopDecel(100000);
466         lower_motor.EnableRequest(false);
467         upper_motor.EnableRequest(false);
468         enabled = false;
469         SerialUdpSendln("Motors disabled");
470       }
471     }
472
473     if (clear_state != last_clear_state) {
474       last_clear_state = clear_state;
475       if (clear_state) {
476         HandleAlerts();
477       }
478     }
479
480     if (lower_ctrl_state != last_lower_ctrl_state) {
481       last_lower_ctrl_state = lower_ctrl_state;
482       if (lower_ctrl_state) {
483         Delay_ms(100);
484         LowerMovePosition(2);  // Move to Position 2 (open) defined in MSP.
485       } else {
486         Delay_ms(100);
487         LowerMovePosition(1);  // Move to Position 1 (closed) defined in MSP.
488       }
489     }
490
491     if (upper_ctrl_up_state != last_upper_ctrl_up_state) {
492       last_upper_ctrl_up_state = upper_ctrl_up_state;
493       if (upper_ctrl_up_state) {
494         SerialUdpSendln("Upper moving up/cylinder closing");
495         Delay_ms(100);
496         UpperMoveAtVelocity(-vel);
497       } else {
498         SerialPort.SendLine("off");
499         UpperMoveAtVelocity(0);
500       }
501     }
502
503     if (upper_ctrl_down_state != last_upper_ctrl_down_state) {
504       last_upper_ctrl_down_state = upper_ctrl_down_state;
505       if (upper_ctrl_down_state) {
506         SerialUdpSendln("Upper moving down/cylinder opening");
507         Delay_ms(100);
508         UpperMoveAtVelocity(vel);
509       } else {
510         UpperMoveAtVelocity(0);
511       }
512     }
513
514
515     delay(10);
516   }
517
518   /*------------------------------------------------------------------------------
519      LowerMovePosition (MC motor)
520
521         Move motor controlling lower cylinder to position number positionNum (defined in MSP)
522         Prints the move status to the USB serial port
523         Returns when HLFB asserts (indicating the motor has reached the commanded
524         position)
```

```
525
526     Parameters:
527        int positionNum  - 1: input A off, 2: input A on
528
529     Returns: True/False depending on whether a valid position was
530        successfully commanded and reached.
531   */
532   bool LowerMovePosition(uint8_t positionNum) {
533     // Check if an alert is currently preventing motion
534     if (lower_motor.StatusReg().bit.AlertsPresent) {
535       SerialUdpSendln("Lower Motor status: 'In Alert'. Move Canceled.");
536       return false;
537     }
538     SerialPort.Send("Moving to position: ");
539     SerialPort.Send(positionNum);
540     SerialPort.SendLine();
541     Udp.Connect(Udp.RemoteIp(), Udp.RemotePort());
542     Udp.PacketWrite("\r\nMoving to position: ");
543     Udp.PacketWrite(itoa(positionNum, inttostr, 10));
544     Udp.PacketWrite("\r\n");
545     Udp.PacketSend();
546
547     switch (positionNum) {
548       // if motor configured in "move to sensor position" mode
549       // case 1:
550       //   lower_motor.MotorInBState(false);
551       //   lower_motor.MotorInAState(true);
552       //   SerialPort.SendLine("Input A On");
553       //   SerialUdpSendln("Lower cylinder closing (moving down) ...");
554       //   break;
555       // case 2:
556       //   lower_motor.MotorInAState(false);
557       //   lower_motor.MotorInBState(true);
558       //   SerialPort.SendLine("Input B On");
559       //   SerialUdpSendln("Lower cylinder opening (moving up) ...");
560       //   break;
561
562       // if motor configured in "move to absolute position" mode
563       case 1:
564         lower_motor.MotorInAState(false); // Sets Input A "off" for position 1
565         SerialPort.SendLine(" (Input A Off)");
566         SerialUdpSendln("Lower cylinder closing (moving down) ...");
567         break;
568       case 2:
569         lower_motor.MotorInAState(true); // Sets Input A "on" for position 2
570         SerialPort.SendLine(" (Input A On)");
571         SerialUdpSendln("Lower cylinder opening (moving up) ...");
572         break;
573       default:
574         // If this case is reached then an incorrect positionNum was entered
575         return false;
576     }
577     // Waits for HLFB to assert (signaling the move has successfully completed)
578     SerialUdpSendln("Moving.. Waiting for HLFB");
579     while (
580       lower_motor.HlfbState() != MotorDriver::HLFB_ASSERTED &&
581       !lower_motor.StatusReg().bit.AlertsPresent
582     ) {
583       if (!enable_button.State()) {
584         break;
585       }
586       // Look for a received packet.
587       packetSize = Udp.PacketParse();
588
589       if (packetSize > 0) {
590         SerialPort.Send("Received packet of size ");
591         SerialPort.Send(packetSize);
592         SerialPort.Send(" bytes. ");
593         SerialPort.Send("Remote IP: ");
594         SerialPort.SendLine(Udp.RemoteIp().StringValue());
595         SerialPort.Send("Remote port: ");
596         SerialPort.SendLine(Udp.RemotePort());
597         // Read the packet.
598         int32_t bytesRead = Udp.PacketRead(packetReceived, MAX_PACKET_LENGTH);
599         SerialPort.Send("Number of bytes read from packet: ");
600         SerialPort.SendLine(bytesRead);
601         SerialPort.Send("Packet contents: ");
602         SerialPort.Send((char *)packetReceived);
603         SerialPort.SendLine();
604
605         char mode = packetReceived[0];
606
607         // Define commands to be read from UDP. will only run case A or B after case 'r' for reset.
```

```
608         // B will take anything (currently 9 digits) after the first char and make it an int.
609         // This could be the #steps for the motor
610         switch (mode) {
611           case 'd':
612             SerialUdpSendln("Aborting moves ...");
613             lower_motor.MoveStopDecel(100000);
614             upper_motor.MoveStopDecel(100000);
615 
616             lower_motor.EnableRequest(false);
617             upper_motor.EnableRequest(false);
618             enabled = false;
619             SerialUdpSendln("Motors disabled");
620             break;
621 
622           case 'a':
623             SerialUdpSendln("Abort move");
624             abort_move = true;
625             lower_motor.MoveStopDecel(100000);
626             Delay_ms(1000);
627             break;
628 
629           case 'S':
630             printStatus();
631             break;
632 
633           default:
634             SerialUdpSendln("No case found");
635             break;
636         }
637       }
638 
639       // clear the packet buffer
640       for(int i=0;i<MAX_PACKET_LENGTH;i++) packetReceived[i] = 0;
641 
642       continue;
643     }
644     // Check if motor alert occurred during move
645     // Clear alert if configured to do so
646     if (lower_motor.StatusReg().bit.AlertsPresent) {
647       SerialUdpSendln("Motor alert detected.");
648       PrintAlerts();
649       if (HANDLE_ALERTS) {
650         HandleAlerts();
651       } else {
652         SerialUdpSendln("Enable automatic fault handling by setting HANDLE_ALERTS to 1.");
653       }
654       SerialUdpSendln("Motion may not have completed as expected. Proceed with caution.");
655       return false;
656     } else {
657       SerialUdpSendln("Move Done");
658 
659       return true;
660     }
661   }
662 
663   /*------------------------------------------------------------------------------
664      UpperMoveDistance
665 
666         Command "distance" number of step pulses away from the current position
667         Prints the move status to the USB serial port
668         Returns when HLFB asserts (indicating the motor has reached the commanded
669         position)
670 
671      Parameters:
672         int distance  - The distance, in step pulses, to move
673 
674      Returns: True/False depending on whether the move was successfully triggered.
675   ------------------------------------------------------------------------------*/
676   bool UpperMoveDistance(int32_t distance) {
677     // Check if a motor alert is currently preventing motion
678     // Clear alert if configured to do so
679     if (upper_motor.StatusReg().bit.AlertsPresent) {
680       SerialUdpSendln("Motor alert detected.");
681       PrintAlerts();
682       if (HANDLE_ALERTS) {
683         HandleAlerts();
684       } else {
685         SerialUdpSendln("Enable automatic alert handling by setting HANDLE_ALERTS to 1.");
686       }
687       SerialUdpSendln("Move canceled.");
688       SerialPort.SendLine();
689       return false;
690     }
```

```
691
692     // check if distance would take cylinder out of movement range
693     if (upper_motor.PositionRefCommanded() + distance < upper_homing_offset) {
694       SerialUdpSendln("WARNING: commanded move less than min position.");
695       SerialUdpSendln("Moving to min position instead.");
696       distance = upper_homing_offset - upper_motor.PositionRefCommanded();
697     } else if (upper_motor.PositionRefCommanded() + distance > cylinder_range) {
698       SerialUdpSendln("WARNING: commanded move greater than max position.");
699       SerialUdpSendln("Moving to max position instead.");
700       distance = cylinder_range - upper_motor.PositionRefCommanded();
701     }
702     SerialPort.Send("Moving distance: ");
703     SerialPort.SendLine(distance);
704     Udp.Connect(Udp.RemoteIp(), Udp.RemotePort());
705     Udp.PacketWrite("\r\nMoving distance: ");
706     Udp.PacketWrite(itoa(distance, inttostr, 10));
707     Udp.PacketWrite("\r\n");
708     Udp.PacketSend();
709     upper_motor.VelMax(vel);
710     // Command the move of incremental distance
711     upper_motor.Move(distance);
712     // Waits for HLFB to assert (signaling the move has successfully completed)
713     SerialUdpSendln("Moving.. Waiting for HLFB");
714     while (
715       (!upper_motor.StepsComplete() || upper_motor.HlfbState() != MotorDriver::HLFB_ASSERTED) &&
716       !upper_motor.StatusReg().bit.AlertsPresent
717     ) {
718       if (!enable_button.State()) {
719         break;
720       }
721       // Look for a received packet.
722       packetSize = Udp.PacketParse();
723
724       if (packetSize > 0) {
725         SerialPort.Send("Received packet of size ");
726         SerialPort.Send(packetSize);
727         SerialPort.Send(" bytes. ");
728         SerialPort.Send("Remote IP: ");
729         SerialPort.SendLine(Udp.RemoteIp().StringValue());
730         SerialPort.Send("Remote port: ");
731         SerialPort.SendLine(Udp.RemotePort());
732         // Read the packet.
733         int32_t bytesRead = Udp.PacketRead(packetReceived, MAX_PACKET_LENGTH);
734         SerialPort.Send("Number of bytes read from packet: ");
735         SerialPort.SendLine(bytesRead);
736         SerialPort.Send("Packet contents: ");
737         SerialPort.Send((char *)packetReceived);
738         SerialPort.SendLine();
739
740         char mode = packetReceived[0];
741
742         // Define commands to be read from UDP. will only run case A or B after case 'r' for reset.
743         // B will take anything (currently 9 digits) after the first char and make it an int.
744         // This could be the #steps for the motor
745         switch (mode) {
746           case 'd':
747             SerialUdpSendln("Aborting moves ...");
748             lower_motor.MoveStopDecel(100000);
749             upper_motor.MoveStopDecel(100000);
750
751             lower_motor.EnableRequest(false);
752             upper_motor.EnableRequest(false);
753             enabled = false;
754             SerialUdpSendln("Motors disabled");
755             break;
756
757           case 'a':
758             SerialUdpSendln("Abort move");
759             abort_move = true;
760             upper_motor.MoveStopDecel(100000);
761             Delay_ms(1000);
762             break;
763
764           case 'S':
765             printStatus();
766             break;
767
768           default:
769             SerialUdpSendln("No case found");
770             break;
771         }
772       }
773
```

```
774       // clear the packet buffer
775       for(int i=0;i<MAX_PACKET_LENGTH;i++) packetReceived[i] = 0;
776
777       continue;
778     }
779
780     // Check if motor alert occurred during move
781     // Clear alert if configured to do so
782     if (upper_motor.StatusReg().bit.AlertsPresent) {
783       SerialUdpSendln("Motor alert detected.");
784       PrintAlerts();
785       if (HANDLE_ALERTS) {
786         HandleAlerts();
787       } else {
788         SerialUdpSendln("Enable automatic fault handling by setting HANDLE_ALERTS to 1.");
789       }
790       SerialUdpSendln("Motion may not have completed as expected. Proceed with caution.");
791       return false;
792     } else {
793       SerialUdpSendln("Move Done");
794       SerialUdpSendln("Current motor position: ");
795       Serial.println(upper_motor.PositionRefCommanded());
796       Udp.Connect(Udp.RemoteIp(), Udp.RemotePort());
797       Udp.PacketWrite(itoa(upper_motor.PositionRefCommanded(), inttostr, 10));
798       Udp.PacketWrite("\r\n");
799       Udp.PacketSend();
800       return true;
801     }
802   }
803
804   /*------------------------------------------------------------------------------
805      UpperMoveAtVelocity
806
807         Command the motor to move at the specified "velocity", in steps/second.
808         Prints the move status to the USB serial port
809
810      Parameters:
811         int velocity  - The velocity, in step steps/sec, to command
812
813      Returns: None
814   ------------------------------------------------------------------------------*/
815   bool UpperMoveAtVelocity(int32_t velocity) {
816     // Check if a motor alert is currently preventing motion
817     // Clear alert if configured to do so
818     if (upper_motor.StatusReg().bit.AlertsPresent) {
819       SerialUdpSendln("Motor alert detected.");
820       PrintAlerts();
821       if (HANDLE_ALERTS) {
822         HandleAlerts();
823       } else {
824         SerialUdpSendln("Enable automatic alert handling by setting HANDLE_ALERTS to 1.");
825       }
826       SerialUdpSendln("Move canceled.");
827       return false;
828     }
829     SerialPort.Send("Commanding velocity: ");
830     SerialPort.SendLine(velocity);
831     Udp.Connect(Udp.RemoteIp(), Udp.RemotePort());
832     Udp.PacketWrite("\r\nCommanding velocity: ");
833     Udp.PacketWrite(itoa(velocity, inttostr, 10));
834     Udp.PacketWrite("\r\n");
835     Udp.PacketSend();
836     // Command the velocity move
837     upper_motor.MoveVelocity(velocity);
838     // Waits for the step command to ramp up/down to the commanded velocity.
839     // This time will depend on your Acceleration Limit.
840     SerialUdpSendln("Ramping to speed...");
841     while (
842       !upper_motor.StatusReg().bit.AtTargetVelocity &&
843       !upper_motor.StatusReg().bit.AlertsPresent
844     ) {
845       if (!enable_button.State()) {
846         break;
847       }
848       continue;
849     }
850     if (upper_motor.StatusReg().bit.AtTargetVelocity) {
851       SerialUdpSendln("At Speed");
852       if (upper_motor.VelocityRefCommanded() == 0) {
853         Serial.print("Current motor position: ");
854        Serial.println(upper_motor.PositionRefCommanded());
855       }
856     }
```

```
857
858     // Check if motor alert occurred during move
859     // Clear alert if configured to do so
860     if (upper_motor.StatusReg().bit.AlertsPresent) {
861       SerialUdpSendln("Motor alert detected.");
862       PrintAlerts();
863       if (HANDLE_ALERTS) {
864         HandleAlerts();
865       } else {
866         SerialUdpSendln("Enable automatic fault handling by setting HANDLE_ALERTS to 1.");
867       }
868       SerialUdpSendln("Motion may not have completed as expected. Proceed with caution.");
869       return false;
870     } else {
871       return true;
872     }
873   }
874
875
876   /*------------------------------------------------------------------------------
877      PrintAlerts
878
879         Prints active alerts.
880
881      Parameters:
882         requires "motor" to be defined as a ClearCore motor connector
883
884      Returns:
885         none
886   ------------------------------------------------------------------------------*/
887   void PrintAlerts() {
888     SerialUdpSendln("Upper motor: ");
889     if (upper_motor.AlertReg().bit.MotionCanceledInAlert) {
890       SerialUdpSendln("- MotionCanceledInAlert ");
891     }
892     if (upper_motor.AlertReg().bit.MotionCanceledPositiveLimit) {
893       SerialUdpSendln("- MotionCanceledPositiveLimit ");
894     }
895     if (upper_motor.AlertReg().bit.MotionCanceledNegativeLimit) {
896       SerialUdpSendln("- MotionCanceledNegativeLimit ");
897     }
898     if (upper_motor.AlertReg().bit.MotionCanceledSensorEStop) {
899       SerialUdpSendln("- MotionCanceledSensorEStop ");
900     }
901     if (upper_motor.AlertReg().bit.MotionCanceledMotorDisabled) {
902       SerialUdpSendln("- MotionCanceledMotorDisabled ");
903     }
904     if (upper_motor.AlertReg().bit.MotorFaulted) {
905       SerialUdpSendln("- MotorFaulted ");
906     }
907
908     SerialUdpSendln("Lower motor: ");
909       if (lower_motor.AlertReg().bit.MotionCanceledInAlert) {
910       SerialUdpSendln("- MotionCanceledInAlert ");
911     }
912     if (lower_motor.AlertReg().bit.MotionCanceledPositiveLimit) {
913       SerialUdpSendln("- MotionCanceledPositiveLimit ");
914     }
915     if (lower_motor.AlertReg().bit.MotionCanceledNegativeLimit) {
916       SerialUdpSendln("- MotionCanceledNegativeLimit ");
917     }
918     if (lower_motor.AlertReg().bit.MotionCanceledSensorEStop) {
919       SerialUdpSendln("- MotionCanceledSensorEStop ");
920     }
921     if (lower_motor.AlertReg().bit.MotionCanceledMotorDisabled) {
922       SerialUdpSendln("- MotionCanceledMotorDisabled ");
923     }
924     if (lower_motor.AlertReg().bit.MotorFaulted) {
925       SerialUdpSendln("- MotorFaulted ");
926     }
927     SerialUdpSendln("");
928   }
929
930   /*------------------------------------------------------------------------------
931      HandleAlerts
932
933         Clears alerts, including motor faults.
934         Faults are cleared by cycling enable to the motor.
935         Alerts are cleared by clearing the ClearCore alert register directly.
936
937      Parameters:
938         requires "motor" to be defined as a ClearCore motor connector
939
```

```
940     Returns:
941        none
942  ------------------------------------------------------------------------------*/
943  void HandleAlerts() {
944    if (upper_motor.AlertReg().bit.MotorFaulted) {
945      // if a motor fault is present, clear it by cycling enable
946      SerialUdpSendln("Upper Motor Faults present. Cycling enable signal to motor to clear faults.");
947      upper_motor.EnableRequest(false);
948      Delay_ms(10);
949      upper_motor.EnableRequest(true);
950    }
951    if (lower_motor.AlertReg().bit.MotorFaulted) {
952      // if a motor fault is present, clear it by cycling enable
953      SerialUdpSendln("Lower Motor Faults present. Cycling enable signal to motor to clear faults.");
954      lower_motor.EnableRequest(false);
955      Delay_ms(10);
956      lower_motor.EnableRequest(true);
957    }
958    // clear alerts
959    SerialUdpSendln("Clearing alerts.");
960    upper_motor.ClearAlerts();
961    lower_motor.ClearAlerts();
962    abort_move = false;
963  }
964
965
966  /*------------------------------------------------------------------------------
967     printStatus
968
969     print Status
970  ------------------------------------------------------------------------------*/
971  void printStatus() {
972    if (lower_motor.EnableRequest()) {
973      if (lower_motor.MotorInAState() && lower_motor.HlfbState() == MotorDriver::HLFB_ASSERTED) {
974        lowerPosition = "Open ";
975      } else if (lower_motor.MotorInAState() && lower_motor.HlfbState() != MotorDriver::HLFB_ASSERTED) {
976        lowerPosition = "Opening ";
977      } else if (!lower_motor.MotorInAState() && lower_motor.HlfbState() != MotorDriver::HLFB_ASSERTED) {
978        lowerPosition = "Closing ";
979      } else if (!lower_motor.MotorInAState() && lower_motor.HlfbState() == MotorDriver::HLFB_ASSERTED) {
980        lowerPosition = "Closed ";
981      }
982    } else {
983      lowerPosition = "Disabled";
984    }
985
986    if (upper_motor.EnableRequest()) {
987      if (upper_HomingSensor.State() && upper_LimitSwitch.State()) {
988        if (upper_motor.HlfbState() == MotorDriver::HLFB_ASSERTED) {
989          upperPosition = "Open ";
990        } else if (upper_motor.VelocityRefCommanded() > 0) {
991          upperPosition = "Opening ";
992        } else if (upper_motor.VelocityRefCommanded() < 0) {
993          upperPosition = "Closing ";
994        }
995      } else if (upper_HomingSensor.State() && !upper_LimitSwitch.State()) {
996        upperPosition = "MaxOpen ";
997      } else if (!upper_HomingSensor.State() && upper_LimitSwitch.State()) {
998        upperPosition = "Closed ";
999      } else {
1000       upperPosition = "???"; //something is wrong
1001     }
1002   } else {
1003     upperPosition = "Disabled";
1004   }
1005
1006
1007   if (upper_ctrl_up_button.State()) {
1008     strcpy(upperCtrlUpState_str, "ON ");
1009   }
1010   if (upper_ctrl_down_button.State()) {
1011     strcpy(upperCtrlDownState_str, "ON ");
1012   }
1013   if (lower_ctrl_button.State()) {
1014     strcpy(lowerCtrlState_str, "ON ");
1015   }
1016   if (clear_button.State()) {
1017     strcpy(clearState_str, "ON ");
1018   }
1019   if (enable_button.State()) {
1020     strcpy(enableState_str, "ON ");
1021   }
1022   if (upper_HomingSensor.State()) {
```

```
1023      strcpy(upperHomeState_str, "ON ");
1024    }
1025    if (upper_LimitSwitch.State()) {
1026      strcpy(upperLimitState_str, "ON ");
1027    }
1028    if (lower_HomingSensor.State()) {
1029      strcpy(lowerHomeState_str, "ON ");
1030    }
1031    if (lower_LimitSwitch.State()) {
1032      strcpy(lowerLimitState_str, "ON ");
1033    }
1034
1035    char *statRepFormat;
1036    statRepFormat = "\r\n"
1037                    "\r\n                  W1 CYLINDER STATUS                "
1038                    "\r\n================================================="
1039                    "\r\n Upper cylinder:           %s (%i steps)"
1040                    "\r\n Lower cylinder:           %s"
1041                    "\r\n"
1042                    "\r\nButton             State | Sensor            State "
1043                    "\r\n-----------------------+-------------------------"
1044                    "\r\n Upper ctrl. up      %s | Upper cyl. home     %s"
1045                    "\r\n Upper ctrl. down    %s | Upper cyl. limit    %s"
1046                    "\r\n Lower ctrl.         %s | "
1047                    "\r\n Clear alerts        %s | Lower cyl. home     %s"
1048                    "\r\n Enable motors       %s | Lower cyl. limit    %s"
1049                    "\r\n"
1050                    "\r\nAlerts"
1051                    "\r\n-------------------------------------------------"
1052                    "\r\n";
1053
1054    snprintf(
1055      statusReport,
1056      1024,
1057      statRepFormat,
1058      upperPosition,
1059      upper_motor.PositionRefCommanded(),
1060      lowerPosition,
1061      upperCtrlUpState_str,
1062      upperHomeState_str,
1063      upperCtrlDownState_str,
1064      upperLimitState_str,
1065      lowerCtrlState_str,
1066      clearState_str,
1067      lowerHomeState_str,
1068      enableState_str,
1069      lowerLimitState_str
1070    );
1071
1072    SerialPort.Send(statusReport);
1073    Udp.Connect(Udp.RemoteIp(), Udp.RemotePort());
1074    Udp.PacketWrite(statusReport);
1075    Udp.PacketSend();
1076    PrintAlerts();
1077  }
1078
1079  void printUpperMoveSettings() {
1080    char moveSettings[1024];
1081    char *moveSettingsFormat = "\r\n"
1082                               "\r\nUpper Cyl. Movement Settings"
1083                               "\r\n-----------------------------------"
1084                               "\r\nPosition increment (steps)   %i"
1085                               "\r\nVelocity (steps/sec)         %i"
1086                               "\r\nMin. position (steps)        %i"
1087                               "\r\nMax. position (steps)        %i"
1088                               "\r\n";
1089    snprintf(moveSettings, 1024, moveSettingsFormat, steps, vel, upper_homing_offset, cylinder_range);
1090
1091    SerialPort.Send(moveSettings);
1092    Udp.Connect(Udp.RemoteIp(), Udp.RemotePort());
1093    Udp.PacketWrite(moveSettings);
1094    Udp.PacketSend();
1095  }
1096
1097  /*------------------------------------------------------------------------------
1098    HomingSensorCallback
1099
1100      Reads the state of the homing sensor and passes the state to the motor.
1101  ------------------------------------------------------------------------------*/
1102  void lower_HomingSensor_Callback() {
1103    // A 1 ms delay is required in order to pass the correct filtered sensor
1104    // state.
1105    Delay_ms(1);
```

```
  lower_motor.MotorInBState(lower_HomingSensor.State());
  if (!lower_HomingSensor.State()) {
    SerialUdpSendln("Lower cylinder home switch triggered.");
  }
  // if motor configured in "move to sensor position" mode
  // lower_motor.MotorInAState(lower_HomingSensor.State());

  // if motor configured in "move to absolute position" mode
  // lower_motor.MoveStopDecel(100000);
}

/*------------------------------------------------------------------------------
   HomingSensorCallback

      Reads the state of the homing sensor and passes the state to the motor.
------------------------------------------------------------------------------*/
void upper_HomingSensor_Callback() {
  // A 1 ms delay is required in order to pass the correct filtered sensor
  // state.
  Delay_ms(1);
  SerialUdpSendln("Upper cylinder home switch triggered. Aborting motion.");
  upper_motor.MoveStopDecel(100000);
}

/*------------------------------------------------------------------------------
   LimitSwitchCallback

      Reads the state of the homing sensor and passes the state to the motor.
*/
void lower_LimitSwitch_Callback() {
  // A 1 ms delay is required in order to pass the correct filtered sensor
  // state.
  Delay_ms(1);
  SerialUdpSendln("Lower cylinder limit switch triggered. Cylinder open.");
  // if motor configured in "move to sensor position" mode
  // lower_motor.MotorInAState(lower_HomingSensor.State());

  // if motor configured in "move to absolute position" mode
  lower_motor.MoveStopDecel(100000);
}

/*------------------------------------------------------------------------------
   LimitSwitchCallback

      Reads the state of the homing sensor and passes the state to the motor.
------------------------------------------------------------------------------*/
void upper_LimitSwitch_Callback() {
  // A 1 ms delay is required in order to pass the correct filtered sensor
  // state.
  Delay_ms(1);
  SerialUdpSendln("Upper cylinder limit switch triggered. Aborting motion.");
  upper_motor.MoveStopDecel(100000);

}

/*------------------------------------------------------------------------------
   SerialUdpSendln

      Sends a string + newline to the serial port and UDP connection.
------------------------------------------------------------------------------*/
void SerialUdpSendln(char *message) {
  SerialPort.SendLine(message);
  Udp.Connect(Udp.RemoteIp(), Udp.RemotePort());
  Udp.PacketWrite(message);
  Udp.PacketWrite("\r\n");
  Udp.PacketSend();
}
```

C. DATA ANALYSIS CODE

The analysis of the sensor data was performed using a Jupyter Notebook, available at the CHARA GitLab cylinders repository as `analysis.ipynb`. The notebook contains routines for reading sensor logs and making plots such as the ones shown in this report.

D. CLEARCORE SPECIFICATIONS AND MANUAL

https://teknic.com/products/io-motion-controller/clcr-4-13/