TOWER-TRACKING HELIOSTAT ARRAY

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Mechanical Engineering

by

Joel Thomas Masters

March 2011

© 2011

Joel Thomas Masters

ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: Tower-Tracking Heliostat Array

AUTHOR: Joel Thomas Masters

DATE SUBMITTED: March 2011

COMMITTEE CHAIR: Saeed Niku, Ph.D., P.E., Professor of Mechanical Engineering

COMMITTEE MEMBER: Pete Schwartz, Ph.D., Professor of Physics

COMMITTEE MEMBER: John Ridgely, Ph.D., Professor of Mechanical Engineering

ABSTRACT

Tower-Tracking Heliostat Array

Joel Thomas Masters

This thesis presents a method of tracking and correcting for the swaying of a central receiver tower in concentrated solar production plants. The method uses a camera with image processing algorithms to detect movement of the center of the tower. A prototype was constructed utilizing a CMOS camera connected to a microcontroller to control the movements of three surrounding heliostats. The prototype uses blob-tracking algorithms to detect and correct for movements of a colored model target. The model was able to detect movements in the tower with average error of 0.32 degrees, and was able to correctly orient the surrounding heliostats to within 1.2 and 2.6 degrees of accuracy while testing indoors and outdoors, respectively.

Table of Contents

List of Figures	vi
List of Tables	vii
Chapter 1: Introduction and Background.....	1
1.1 Introduction	1
1.2 Objective	3
1.3 Background	3
1.3.1 What is a heliostat?	3
1.3.2 What a heliostat is used for.....	3
1.3.3 How a Heliostat Works	7
1.3.4 Image Sensors	8
1.3.5 Finite State Machines.....	9
1.4 Current State of the Art	10
1.4.1 Methods for Determining the Position of the Sun	10
1.4.2 Methods for Determining the Position of the Target	13
1.4.3 Calibration Methods.....	14
Chapter 2: Prototype	17
2.1 Model	17
2.1.1 Hardware.....	19
2.1.2 Software	23
2.1.3 Transformation Relationships.....	44
Chapter 3: Experiments and Results	49
3.1 Calibrating the Camera.....	49
3.1.2 White Balance, Gain Correction, and Exposure Control	49
3.1.3 Determining the Pixel to Distance Relationship.....	50
3.2 Calibrating the Servos	54
3.3 Determining the Position of the Target	56
3.4 Accuracy of Prototype.....	60
3.5 Depth Study	62
3.6 Correcting for Movements of the Base	63
Chapter 4: Conclusions	67
4.1 Conclusions	67
4.2 Future Work	67
References	69
Appendix.....	71
Appendix A: Programming the Blackfin	71
Appendix B: Servo Calibration Plots	72
Appendix C: Program Code	75

List of Figures

Figure 1: PS20 Solar Collection Power Plant by Solúcar. [9]	4
Figure 2: Sierra SunTower created by eSolar in Lancaster, California. [1]	4
Figure 3: Rooftop Heliostats in Battery Park City, New York. [10]	5
Figure 4: Heliostat on the roof of the Haus der Forschung in Vienna redirecting light into a series of fiber optic cables. [11]	5
Figure 5: Practical Solar 1x1m Small-Scale Heliostat. [12]	6
Figure 6: Scheffler Mirror used for solar cooking. [14]	6
Figure 7: Coelestat alignment throughout the day [15]	8
Figure 8: Image-based sun tracking [19]	11
Figure 9: Sun Sensor used on Heliotrack's 1m by 1m heliostat. [20]	12
Figure 10: Photosensor array used for determining the position of the sun relative to the mirror. [6].....	12
Figure 11: Detecting the tower using infrared receivers. [5]	13
Figure 12: Position correction based on reflected solar image on target. [7]	14
Figure 13: Comparing reflected images with four different cameras surrounding the target. [3].....	15
Figure 14: Calibration Method proposed by Ulmer <i>et al.</i> using a projected pattern to determine the optical quality of heliostats [22].....	16
Figure 15: Constructed Model	17
Figure 16: Model Functionality Overview	18
Figure 17: SRV-1 Blackfin Camera Module	19
Figure 18: Servo pan/tilt mechanism	20
Figure 19: Electrical Wiring Diagram	22
Figure 20: Software Overview	24
Figure 21: Mastermind State Transition Diagram	27
Figure 22: Image Capture Task state transition diagram	30
Figure 23: User Interface State Transition Diagram.....	38
Figure 24: Move Mirror X State Transition Diagram.....	40
Figure 25: Outside tower source image	41
Figure 26: Outside tower with blob detection enabled	41
Figure 27: Sun source image.....	42
Figure 28: Sun image with blob detection enabled.....	42
Figure 29: Coordinate references.....	44
Figure 30: Bisecting vector representation	47
Figure 31: Setup of camera calibration. The square checkerboard pattern is placed at a depth of 3 inches from the camera.	51
Figure 32: Camera's view of the checkerboard in Figure 27 at 640x480 resolution, with pixel lengths specified for several squares. Each square is 0.5 inches by 0.5 inches.	52
Figure 33: Distortion due to camera lens. The further away from the center the pixel is, the more deviant it is from actual position.	52
Figure 34: Linear portion of camera lens distortion with up to 14 degrees of linearity. .	53
Figure 35: Camera Azimuth Servo Calibration (Servo 1)	54

Figure 36: Camera Altitude Servo Calibration (Servo 2)	55
Figure 37: Servo Calibration Setup. A protractor is attached to the output shaft of the servo.	55
Figure 38: Target positioned at a depth of 24 inches and 0 inches horizontal offset. Each square is 6x6 inches.	56
Figure 39: Average angular offset error.....	57
Figure 40: Azimuth angular displacements at a depth of 24 inches	58
Figure 41: Azimuth angular displacements at a depth of 36 inches	58
Figure 42: Azimuth angular displacements at a depth of 48 inches	59
Figure 43: Average Vertical displacement of the tower with respect to the camera	59
Figure 44: Mirror Number Designation	60
Figure 45: Reflected Solar Images at 8:45am.....	60
Figure 46: Reflected Solar Images at 9:05am.....	61
Figure 47: Reflected Solar Images at 9:35am.....	61
Figure 48: Radial offsets of the reflected solar images at different times of the day	61
Figure 49: Radial offsets of reflected images on target	62
Figure 50: Angular errors associated with depth offset	63
Figure 51: Base tilted left.....	64
Figure 52: Base tilted forward	64
Figure 53: Mirror 1 angular error after correcting for base tilt.....	65
Figure 54: Mirror 2 angular error after correcting for base tilt.....	65
Figure 55: Mirror 3 angular error after correcting for base tilt.....	66

List of Tables

Table 1: GPS Variable States.....	36
Table 2: Number of pixels per degree offset	54

Chapter 1: Introduction and Background

1.1 Introduction

In the search for clean, renewable energy sources, companies have turned toward concentrated solar production as a way to reduce harmful emissions. One of the disadvantages of concentrated solar power is that it is not yet as cost-effective as most alternatives, specifically the burning of fossil fuels. With technology progressing constantly, and new companies [1][2] emerging with modular, easily produced and assembled structures, the cost gap between concentrated solar and the alternatives is closing.

It is known that the sun can be tracked accurately [3], but little effort has been placed into tracking and correcting for movements in the receiving tower. While movements in the tower may be slight (estimated to be 0.5% of the tower height [4]), they can cause some reflected sunlight from the heliostats to miss the target. Sunlight that misses the target is a loss in energy conversion, and decreases the overall efficiency of the system. The maximum efficiency of concentrated solar power systems cannot exceed the Carnot Efficiency, which is based on the maximum temperature difference between the input reservoir and the rejection reservoir. If mirrors are aiming incorrectly and no longer focusing completely on the tower, the amount of energy into the system is not at the maximum and the working fluid will not get as hot, resulting in less usable energy out of the system. Thus, it is desirable to reflect as much sunlight as possible onto the receiving tower.

One way to decrease the cost of harnessing solar power is to increase its efficiency for a given amount of reflective surface. The laws of physics govern the

amount of usable energy that can be harnessed based on a certain amount of energy input, and cannot be changed. However, tower-tracking can increase the energy into the system, resulting in more energy out of the system for the same reflective surface area.

Methods have been proposed to account for movements in the tower, such as presented in Section 1.4.2 [5][6], which take sensor-based approaches to detect any displacement. However, these sensor-based methods would require a redesign of current heliostats to account for sensor arrays and allow light to pass through the mirror centers. Other methods, such as presented in Section 1.4.3[3][7], have used cameras to account for any offsets of the heliostat accrued through operation by comparing reflected solar images to desired solar images. These methods require cameras resistant to high-heat, and result in a loss in energy production during calibration times.

This thesis prototypes a method of tracking the tower and correcting the orientations of several surrounding heliostats using a small CMOS camera and a microcontroller to determine the locations of the tower and light source and orient three surrounding heliostats. The camera and heliostats are moved via servos in pan/tilt orientations. The heliostats and camera are connected through a rigid structure, such that the location of each heliostat with respect to the camera is known. This method has a two-fold purpose: aside from detecting shifts in the tower itself, the cameras will be able to detect and correct for movements of the floor. This could be advantageous for up-and-coming rooftop heliostat projects, where the tower and rooftop can shift due to environmental factors such as wind and heat loads.

1.2 Objective

The objective of this thesis is to develop a prototype of a method of tracking and correcting for tower sway of a concentrated solar collection tower in order to maximize energy gains.

1.3 Background

1.3.1 What is a heliostat?

A heliostat is a mirror that tracks the sun and reflects its light to a constant position throughout the day. In this report, the position will be referred to as the target, or in the case of energy production, the tower.

1.3.2 What a heliostat is used for

Heliostats are most commonly used in Concentrated Solar Power (CSP) plants, whereby large amounts of sunlight are reflected onto a central tower to convert heat into energy. An example of a CSP plant is the PS10 project in Spain. PS10 is an 11 MW Solar Thermal Power Plant that generates 24.3GW-h of energy per year [8]. The system works by heating water to saturated steam at 250°C and 40 bar at the top of the tower and sending it back down and through a steam turbine that converts the steam into electricity. This system uses 624 heliostats each with 120m² of reflective surface that are curved to concentrate the sun rays onto the collection tower. The collection area of the tower is located 115m up and is 14m wide, and provides approximately 200m² of heat-exchange surface area with the water.

Figure 1: PS20 Solar Collection Power Plant by Solúcar. [9]

Another CSP plant is eSolar's 5 MW Sierra SunTower in Lancaster, California. The Sierra SunTower contains two radiation collection towers that stand nearly 65m tall. This system uses 24,000 small (approximately 1.5m by 1m), flat mirrors to focus the power of nearly 20,000 suns onto two collection towers. The process for converting radiation into energy is the same as the PS10, where water is pumped up to the solar collection tower, the pumped back as saturated steam and converted into electricity through a steam turbine.

Figure 2: Sierra SunTower created by eSolar in Lancaster, California. [1]

Heliostats are also used for natural daylighting, reflecting sunlight into areas that would normally receive no sun. One example is the Teardrop Park, located in Battery

Park City in New York, which is a park surrounded by tall buildings that obstruct the sunlight. Heliostats mounted on the rooftop of a nearby building redirect light onto the park to provide the sunlight that the plants need to survive.

Figure 3: Rooftop Heliostats in Battery Park City, New York. [10]

Another application is the Haus der Forschung in Vienna. This building uses a heliostat to redirect sunlight into a series of fiber optic cables and down into the foyer of the building. The result is a “cold-light” that is transmitted from the rooftop down to the foyer. The term “cold-light” is used to describe light that remains cold to the touch and/or emitted by a process other than incandescence.

Figure 4: Heliostat on the roof of the Haus der Forschung in Vienna redirecting light into a series of fiber optic cables. [11]

Heliostats are also used on a smaller scale to provide natural lighting to houses. One example is a 1m by 1m heliostat developed by Practical Solar [12]. The heliostat

can redirect sunlight into windows to provide natural lighting and thereby cut down energy consumption for lighting.

Figure 5: Practical Solar 1x1m Small-Scale Heliostat. [12]

Heliostats can also be used for heating, either a space in a building or heating a stove in the case of a solar cooker. The sun emits radiation to the Earth at approximately 1 kW/m^2 [13]. Scheffler mirrors are heliostats commonly used to concentrate the sun's radiation onto a stovetop for cooking in poorer countries. The 1x1m mirrors by Practical Solar are advertised as being able to heat up rooms in buildings and houses and melt snow off of roofs.

Figure 6: Scheffler Mirror used for solar cooking. [14]

1.3.3 How a Heliostat Works

A heliostat consistently reflects light in one direction by positioning itself at the bisecting angle between the angle to the sun and angle to the target. Most heliostats use computer-controlled tracking to determine the bisecting angle, where the known location and orientation of the heliostat are run through astronomical equations to determine the position of the sun relative to the heliostat, with the target assumed to be at a known location. Other heliostats use sensors such as photoresistors to determine the orientation of the sun. Some heliostats use no tracking at all, and are moved either manually or rotated based solely on the time of day.

Heliostats are most commonly mounted on dual axes that allow rotation in the altitude and azimuth orientations. In this case, the vectors to the sun and to the tower are bisected to determine the correct orientation to reflect sunlight onto the target. The heliostats are moved via motors coupled to the axes of rotation, with rotational position feedback such as resolvers, or by using actuators with inherent position control such as stepper motors, to keep track of their orientations. Some heliostats are rotated on gimbals to achieve desired altitude and azimuth angles. In these cases, the mirror is mounted on a semi-circle to allow rotation in the altitude direction. The base moves about a pivot to allow rotation in the azimuth direction (see U.S. Patent No. 4,2119,22 in Section 1.4.2). Other heliostats rotate on only one axis throughout the day. In this case, the mirrors are typically moved by clock-like mechanisms, moving through a certain angle each hour of the day. Also called coelostats, some mirrors are mounted aligned with the polar axis and rotate at half the solar rate on only one axis. Coelostats are disadvantageous in that the

projected size of mirror is diminished as the solar declination angle becomes more extreme, and need to be shifted daily due to daily declination changes. [15]

Figure 7: Coelestat alignment throughout the day [15]

1.3.4 Image Sensors

There are two main types of image sensors: charge-coupled devices (CCD) and complementary metal oxide semiconductor (CMOS). A charge-coupled device is commonly used in digital cameras. It is a pixelated metal oxide semiconductor that accumulates electrons on its surface based on the intensity of light. When the exposure to light is complete, the chip sends the charge on each pixel to an output structure that sequentially converts the charge at each pixel into a voltage. Once all pixels have been converted, the voltage levels are sent off of the chip.

A complementary metal oxide semiconductor is another device commonly used in digital cameras. Like a CCD, it is a pixelated metal oxide semiconductor that accumulates charge based on the intensity of the local light. However, the charge on each pixel is converted into a voltage on the pixel itself, instead of being transferred to an output structure. Instead of having a single amplifier at the output like a CCD, a CMOS sensor has the same number of amplifiers as pixels.

Both CMOS and CCD sensors capture color by placing red, green, and blue color filters in front of the detector pixels, decreasing the overall spatial resolution of the sensor. The Bayer-pattern is the most commonly used color filter matrix, which combines one red, two green, and one blue pixel to create one full-color pixel, mimicking the perception of the human eye [16].

CMOS sensors are typically known for being smaller, and consuming less power than their CCD counterparts. On the other hand, CCDs are known for being more sensitive and precise, and more suited for industrial, medical, and scientific applications [17].

1.3.5 Finite State Machines

A finite state machine is a software architecture that breaks functions up into tasks, and sub-functions into states, in order to perform multiple functions at the same time. This is known as cooperative multitasking. In cooperative multitasking, each task is called, in order, from a main function. Each time a task is called, it executes one state. The state that each task is in is dependent on the particular task, and what function it is trying to perform. Some tasks follow a linear progression through the states, whereas others have a hub-like state to which the task keeps coming back. One way to break up tasks is to separate them based on hardware [18]. Tasks can also be broken up based on function. For this thesis, the camera, servos, and user interface are all broken up into separate tasks, with a mastermind task to control the entire flow of the program (see Section 2.1.2).

Associated with finite state machines are state transition diagrams. A state transition diagram is a visual representation of the path that the program takes through a

particular task. Each task has its own state transition diagram, explaining what conditions cause the program to move to different states (see Section 2.1.2). The ovals in the diagram represent the different states. The arrows designate paths along which the program can move. The conditions next to the arrow and above or to the left of the forward slash represent conditions that could cause the program to move along that path to the next state. The conditions below or to the right of the forward slash represent the conditions that are set based on the transition from one state to the next.

1.4 Current State of the Art

Currently, there are several different methods for tracking the sun, but few for tracking the target. In most cases, the target is assumed fixed and unable to move. Several large-scale CSP plants also employ methods of calibrating the heliostats to correct for offsets caused by accrual of errors during operation.

1.4.1 Methods for Determining the Position of the Sun

The most common type of solar tracking is making use of astronomical equations to predict the location of the sun. These equations require the global position of the heliostat (longitude and latitude) and time of day to determine the altitude and azimuth angles that the sun makes with respect to the heliostat. Equations have been developed with accuracies of 0.1 mrad [3]. Typically for large-scale power plants, a central computer calculates the altitude and azimuth sun angles for each heliostat, knowing the location and heading of each heliostat.

One method of tracking the sun uses an image-based control scheme, as presented in U.S. Patent No. 12/497,385 [19]. In this patent, a camera attached to each heliostat pointed in the direction of the mirror's normal captures an image of the sun

every 15 seconds and compares the location of the sun in the image with an estimated location of the receiver tower. The program begins each day when light intensity is detected above a predetermined threshold. The estimated location of the tower is determined at the beginning of the day by setting the gain and exposure settings on the camera to a level that allows the camera to pick out distinguishing features on the receiver and stores the location of the tower into memory. Then, the camera gain and exposure settings are changed to allow the sun to be captured without over-saturating the image. The program tracks the sun throughout the day and moves the mirror to have the sun blob and receiver location antipodal with respect to the center of the image.

FIG. 5

Figure 8: Image-based sun tracking [19]

Another method of tracking the sun is through the use of sun sensors, such as photoresistors, which change resistance based on the amount of light incident. Photoresistors have been used in several different orientations, such as proposed in Heliotrack's 1x1m heliostat [20], and Patent No. 4,211,922 [6]. In Heliotrack's design, four photoresistors are mounted surrounding a four-walled post. This post is designed to shade the sensors based on where the sun is located. The photoresistors and post are attached to a dual-axis actuator that scans the sky for the most intense light source. The

sensor scans the sky until all four sensors receive the same intensity of light, signaling that the post is pointed directly at the sun. When the light source is found, the angles the sun makes with respect to the sensor can be transformed into angles of the sun with respect to the mirror.

Figure 9: Sun Sensor used on Heliotrack's 1m by 1m heliostat. [20]

U.S. Patent No. 4,211,922 [6] proposed a method of tracking both the sun and the target by using a photosensor array receded from the face of the mirror. A lens allows a sliver of the sun's image to pass through the heliostat and onto a bar of photosensors. The relative angle of the sun to the heliostat can be estimated based on which photosensors are detecting light.

Figure 10: Photosensor array used for determining the position of the sun relative to the mirror. [6]

1.4.2 Methods for Determining the Position of the Target

The most common type of specifying the position of the target is to pre-determine it relative to the heliostats through measurement and layout. However, methods have been proposed for locating and tracking the tower. Patent No. 4,013,885 [5] describes a method using infrared detectors to determine the location of both the tower and the sun relative to a particular heliostat. The IR detector is recessed from the center of the heliostat with a lens in the center that allows IR waves to pass through the mirror. The solar energy receiver typically gets between 1500°F and 2500°F and the patent claims the infrared energy can be sensed even at large distances from the tower. Patent No. 4,211,922 described in Section 1.4.1 uses a similar method for detecting the tower, allowing the reflected sunlight from the tower to pass through the center of the mirror and be sensed by photoresistors recessed from the mirror.

Figure 11: Detecting the tower using infrared receivers. [5]

Another method proposed in Patent No. 5,862,799 [21] uses a small reflector atop the solar collection tower to reflect a small portion of light back into a receiver. The receiver compares the position of the reflected image to the expected position, and can determine offsets.

1.4.3 Calibration Methods

A method proposed by Berenguel, *et al.* [7] uses a CCD camera to view the sun's reflected image from a heliostat onto the target, and compares the position to the expected position of the sun. Each heliostat is calibrated individually based on a single camera. The program uses threshold detection techniques to determine the centroid of the sun's reflected image. The location of the reflected image is compared with the desired location of the reflected image (in the center of the frame) and the heliostat is controlled appropriately.

Figure 12: Position correction based on reflected solar image on target. [7]

Kribus *et al.* [3] proposed a method of closing the control loop of heliostats by using four CCD cameras located near the target. The cameras were placed at a cross pattern around the target; one on top, one on bottom, one left, and one right of the target. When the light was reflected onto the target, each camera captured an image. These images were then compared based on intensity of light, and the heliostat was corrected accordingly. The cameras were aimed at one specific heliostat.

Figure 13: Comparing reflected images with four different cameras surrounding the target. [3]

Another method of calibration used for determining the optical quality of the heliostats was proposed in “Automated high resolution measurement of heliostat slope errors” [22]. Ulmer *et al.* projected a stripe pattern onto the central receiver tower of a heliostat field and used a MATLAB script to detect canting and other surface defects in the heliostats. The heliostats were positioned to reflect the stripe pattern from the central receiver tower into a camera located near the tower. The camera captured images of each heliostat, and determined defects by analyzing the distortion of the pattern. It took roughly one minute to analyze each mirror, allowing a full field to be analyzed in one night. Figure 14 shows the stripe pattern on the receiver tower (above) the pattern reflected by a heliostat (below).

Figure 14: Calibration Method proposed by Ulmer *et al.* using a projected pattern to determine the optical quality of heliostats [22]

Chapter 2: Prototype

2.1 Model

The prototype developed uses a camera mounted on a pan/tilt mechanism driven by servos to determine the location of a color-coded target and convert the corresponding angles to three adjacent mirrors, each attached to a different pan/tilt mechanism in the same orientation as the camera. The camera and mirrors are connected through a rigid structure made from PVC, so that the location and orientation of each mirror is known relative to the camera.

Figure 15: Constructed Model

Figure 16 explains the overall flow of events of the prototype. The model is turned on, initialized, and prompts the user for several inputs. Once valid inputs have been recorded, the camera locates the tower and the sun using blob-tracking algorithms, and then moves back to the known location of the tower while moving the attached mirrors to

the desired orientations calculated based on the locations of the Sun and tower. From there, the program waits for the user to specify whether to relocate the tower or relocate the Sun, and recalculates the desired orientations of each mirror based on the new information.

Figure 16: Model Functionality Overview

2.1.1 Hardware

Camera and Microprocessor

The camera and microprocessor chosen for this model was part of a package developed by Surveyor Corporation [23], called the SRV-1 Blackfin Camera, which has an Omnivision OV7725 CMOS imaging sensor attached to an Analog Devices Blackfin 537 microprocessor. For this model, all image processing is done onboard the Blackfin microprocessor.

Figure 17: SRV-1 Blackfin Camera Module

This specific package was chosen because of its supporting firmware that handles image capture, storing, and processing. Rather than using a high-resolution camera attached to a full-sized computer, this package was chosen to test the limitations of small CMOS cameras, and determine whether they are sufficient to detect tower sway. The small package and low power consumption of this model used could have significant advantages from a portability standpoint, such as heliostat arrays set up on military base

camps. In this case, the camera can be used to automate the setup of the heliostat array, where the location of the target and sun are automatically found, resulting in much shorter setup times when compared with having to manually determine the location of the target with respect to the heliostat.

Drive Mechanism

In order to position the mirrors and the camera, Hitec HS-332 servos are used in pan/tilt orientations to simulate the dual-axis movement of most production heliostats. The calibration curves for each servo are attached in Appendix B. These servos were found to have a maximum step size of roughly 0.3 degrees, and as small as 0.1 degrees, limited by the mechanical properties of the servos.

Figure 18: Servo pan/tilt mechanism

Support Electronics

All of the support electronics are attached to a general purpose printed circuit board via male/female header pairs. Each of the support electronics has an associated PCB, with 0.1" male breakout headers for easy access to each pin. The electronics include the SRV-1 Blackfin Camera assembly, a logic level shifter, a serial servo controller, a USB-to-UART serial converter, and two step-down voltage regulators.

The SRV-1 Blackfin Camera assembly is programmed using a CP2102 USB-to-UART serial converter that outputs digital logic at 3.3V. For reference in programming the microprocessor, see Appendix A. The USB-to-UART serial converter allows communication between the microcontroller and an external computer.

The logic level shifter is used to convert signals exiting the Blackfin module from 3.3V logic to 5V logic required by the Pololu Micro Serial Servo Controller. The servo controller is responsible for generating the proper pulse-widths for controlling the angular positions of the servos. This servo controller takes inputs of 1000 to 5000, which correspond to a range of approximately 200° degrees of motion.

The two voltage regulators are used to step down voltage from 7.2V battery packs to the necessary 3.3V and 5V requirements of the circuit. The 3.3V regulator steps down the voltage for powering the microprocessor and the low-voltage side of the level shifter, with a maximum output current of 1.5A. The 5V regulator is used to power the servos, and has a maximum output current of roughly 5A. The 5V source for powering the servo controller and the high-voltage side of the level shifter comes from the USB-to-UART serial converter. The wiring diagram can be seen in Figure 19.

Figure 19: Electrical Wiring Diagram

Construction Materials

The support structure is made out of 1-inch PVC with threaded end supports on the top and bottom. These supports provide a pretty flat surface for mounting the servos and for sitting on the ground. The servo mounts were custom designed and printed on a rapid prototyping machine, made out of ABS plastic. The electronics are attached to a wooden support that rests on the PVC structure.

PVC was chosen as the primary material because it is easy to cut and easily pieced together, with many modular components to comprise the shape desired for this project. Also, it allows quick adjustment of the structure's geometry for testing different cases, such as the mirrors being at locations different from originally specified.

PVC was also chosen for the construction of the tower, using the same design philosophy as above. The target on the tower is a printed square with RGB color value of [201, 62, 28] for red, green, and blue, respectively, which represents a red-orange color. This color was chosen in an attempt to minimize background noise while outdoors. The printed square is fixed to a clear plastic sheet that it is in turn fixed to the tower. The total target is a 6- by 6-inch square, with total area of 36 square inches.

The servo mounts and camera mount were designed in SolidWorks and printed courtesy of the Mechanical Engineering Department at Cal Poly San Luis Obispo. The servo mounts connect the servos to the metal supports connected to the PVC. The camera mount affixes the camera to the servo mount.

The mirrors are 6-inch by 6-inch flat mirrors. These mirrors were chosen based on their availability and low cost. The mirrors are attached to the servo mounts via double-sided foam tape, which provides sufficient adhesion to prevent the mirrors from falling off.

2.1.2 Software

The purpose of this model is to demonstrate the feasibility of detecting small motions (sub-degree) of the tower, and accurately calculate the desired corrections for the surrounding heliostats.

Figure 20: Software Overview

The software is programmed in C. The software architecture was designed as a variant of a finite state machine, where processes are broken into tasks, and sub-processes are broken into states. The main tasks are broken up into following sections.

Mastermind Task

The Mastermind task is the task that controls the flow of all other tasks in the program. Different events, such as tracking the sun and tracking the tower, occur through the course of the program; the Mastermind is in charge of ensuring the events occur when they are supposed to.

Upon startup, the Mastermind task calls a setup function which initializes all important variables and moves the servos to 90 degrees straight ahead in the azimuth direction, and roughly 10 degrees in the altitude direction to prevent servo buzz. Then, the Mastermind task waits until the User Interface task has received the required inputs from the user, and tells the program to start searching for the tower. Once the tower is found, the Mastermind task tells the program to search for either the Sun or a light source, depending on inputs from the user. Once the Sun or light source is found, the Mastermind tells the program to track the tower.

- **State 0:** This is the initialization state, which sets appropriate initial values for variables and moves the servos to their ready positions.
- **State 1:** This state waits for a command from the User Interface task to signal that the user has input valid parameters. Upon the signal, the task tells the program to move to the estimate of the tower location based on user inputs.
- **State 2:** State 2 does nothing until receiving a signal from the Move Camera task that the tower has been located, at which time it tells the program to angle the camera toward the known location of the Sun or light source. If the option has been selected to scan for the sun, the program will instead increase the altitude of the camera until a bright spot is found in the sky.
- **State 3:** State 3 does nothing until receiving a signal from the Move Camera task that the sun has been centered within the wide threshold. When this occurs, auto white balancing, auto gain control, and auto exposure control are turned back on. Once the sun has been located, the task tells the program to move to the known location of the tower, turn off AWB, AGC, and AEC, and calculate the global

positions of the tower and light source (if the option is selected). The task then proceeds to State 4.

- **State 4:** State 4 does nothing until the location of the tower has again been determined, and then transitions to State 5.
- **State 5:** State 5 does nothing until the location of the tower and Sun or light source have been calculated, and each mirror has moved to its desired location. Once the mirrors are in the desired orientations, the program prompts the user for three options: re-find the tower, re-find the Sun, or exit the program. If the option to re-find the tower is selected, the task transitions back to State 4, AWB, AGC, and AEC are turned off, and the move_to_tower flag is set high. If the option to re-find the Sun is selected, AWB, AGC, and AEC are turned on upon transition to State 6, and the move_to_sun flag is set high.
- **State 6:** State 6 does nothing until the Move Camera task signals that the Sun has been located, and then transitions back to State 5. The calc_pos flag is set high upon transition, telling the program to calculate the new positions of the Sun and tower, and move the heliostats accordingly.

Figure 21: Mastermind State Transition Diagram

Image Capture Task

The Image Capture task is the task responsible for capturing and processing images from the camera. It first configures the camera to the appropriate resolution, in this case 320x240 pixels, and then turns off automatic white balancing, automatic gain control, and automatic exposure control and stores the camera register values in an array. The camera register values are stored in an array so that they can be sent back to the camera when searching for the target again. The camera registers contain information regarding the amount of gain control and values for different channels of white balancing. If the same register settings are not used, then there could be inconsistencies when trying to locate the tower once again.

After initialization, the camera captures an image and calls the blob detection algorithm twice. The first time it scans the image for pixels corresponding to the color ranges of the target; the second time through it scans for white pixels corresponding to the Sun or light source. Flags are set based upon the blobs detected (whether red, white, both, or no blobs). Once the image has been processed, the Image Capture task sets the image process flag high to allow the Move Camera task to advance. The Image Capture task then waits for the Move Camera task to set the capture flag low to capture another image.

- **State 0:** State 0 is the initialization task. It waits for a command from the User Interface task signaling that valid parameters have been input by the user. Once the command is received, the state sets the camera resolution, prompts the user to place a white paper in front of the camera to calibrate white balancing, and waits until the user hits a key before continuing. When a key is entered, automatic

white balancing, automatic gain control, and automatic exposure control are disabled, and the values for the camera registers are stored into an array. After the camera registers are stored in an array, the task transitions to State 1.

- **State 1:** State 1 captures an image from the camera upon entering the state, but waits until the capture flag goes low for subsequent image captures. After capturing an image, the task proceeds to State 2.
- **State 2:** This state processes the image, looking for blobs corresponding to the tower and Sun or light source. The task transitions to different states depending on one of three conditions: only the color corresponding to the tower is found; only the color corresponding to the Sun or light source is found; or both colors are found. If only red is found, corresponding to the tower, the task transitions to State 3. If only white is found, corresponding to the Sun or light source, then the task transitions to State 4. If both red and white are found, the task transitions to State 5. If neither color is found, the task transitions back to State 1, and signals that the image processing is complete.
- **State 3:** This state sets flags signaling that the tower has been found in the image and not the Sun or light source, signals that the image processing is complete, and tells the task to transition back to State 1 to get ready to capture another image.
- **State 4:** This state sets flags signaling that the Sun or light source has been found and not the tower, signals that the image processing is complete, and tells the task to transition back to State 1.

- **State 5:** This state sets flags signaling that both the tower and Sun or light source have been detected, signals that the image processing is complete, and tells the task to transition back to State 1.

Figure 22: Image Capture Task state transition diagram

Move Camera Task

The Move Camera task's function is to move the camera to find the target and the Sun or light source, calculate the vertical and horizontal offsets of each, and signal when to move the heliostats.

- **State 0:** State 0 is the initialization state which sets the ranges for centering the target in the camera's view (as described in the Track Color function).
- **State 1:** State 1 waits until the signal from the Image Capture task that an image has been processed, and then determines where it should branch based on the objects detected in the image. If only the tower is found, the task transitions to State 2. If only the Sun or light source is found, the task transitions to State 4. If both the tower and Sun or light source are found, the task transitions to State 14. If neither the tower or Sun or light source is found, the task transitions to State 6.
- **State 2:** State 2 is the case where only the tower was found in captured image. This state determines where to transition based on conditions set by the Mastermind task. If the tower is being tracked, the task will move to State 3. If the tower needs to be found, the task will move to State 9. If the camera is supposed to move to the last known location of the tower, the task will transition to State 7. Likewise, if the tower is detected in the frame but the program needs to find the sun, the task will transition to State 10. If the Sun needs to be tracked, the task will transition to State 5. Finally, the task will transition to State 8 if the camera needs to move to the last known location of the sun.
- **State 3:** This state locates the center of the tower. The Track Color function is called to determine the center of the tower in the camera's image. If the blob

center is not within range of the image center, the camera servos are moved to center it. If the tower is centered in the image, the task sets a flag signaling that the tower has been located, and the task transitions to State 12. The capture flag is set low each time through the state, except when the center of the tower is in the center of the image.

- **State 4:** The task moves into State 4 if only the Sun has been found in the captured image. The transitions out of the state are the same as for State 2.
- **State 5:** This state locates the center of the Sun or light source. This state functions the same as State 3, but the Track Color function is called to detect white in the image. If the center of the Sun or light source is not within range of the center of the image, the task moves the camera servos appropriately, and the capture flag is set low. If the center of the Sun or light source is within range of the center of the image, the task sets a flag signaling that the Sun or light source has been located.
- **State 6:** The task transitions to State 6 when both the tower and the Sun or light source are found within the image. The transitions out of the state are the same as States 2 and 4.
- **State 7:** This state moves the camera to the known location of the tower. The task transitions to State 16 once the desired location is reached.
- **State 8:** State 8 moves the camera to the last known location of the Sun or light source. The task transitions to State 15 once the desired location is reached.

- **State 9:** This state moves the camera to scan for the tower. The camera's altitude is increased until the tower is detected in the image. When the tower is detected, the task tells the program to track the tower, and transitions back to State 1.
- **State 10:** State 10 is the same as State 9 except it scans for the Sun or light source. When the Sun or light source is detected, the task tells the program to track the Sun or light source, and transitions back to State 1.
- **State 11:** The task transitions into State 11 from State 5, after the Sun or light source has been centered in the camera's image. The azimuth and altitude angles of the camera servos are stored, and the task transitions into State 13 if the calculate position flag is set high, and into State 1 otherwise.
- **State 12:** State 12 is reached on transition from State 3 after the tower has been centered in the camera's image. If the calculate position flag is high, the task will transition to State 13. Otherwise, it transitions back to State 1 and set the capture flag low.
- **State 13:** State 13 is where the global positions of the tower and light source (if the option is selected) are calculated. The task sets the move_mirror_x flags high and brings the capture flag low upon transition out of the state to State 1.
- **State 14:** The task transitions into this state when both the tower and Sun or light source are found in the image. The transitions out of this state are the same as State 2.
- **State 15:** State 15 tells the program to track the sun or light source. The capture flag is set low upon transition back to State 1.

- **State 16:** State 16 tells the program to track the tower. The capture flag is set low upon transition back to State 1.

User Interface Task

The function of the User Interface task is to prompt the user for several inputs. The first input is whether the program will track the Sun or track a light source. If the Sun is being tracked, the program will present the user with the option to use GPS coordinates to estimate the location of the Sun, or scan for the sun by increasing the altitude of the camera. If the GPS option is selected, the program prompts the user to enter in the longitude, latitude, angle with respect to north, month, day, year, and time of day. The horizontal and vertical offsets of the Sun will not be calculated in the global frame. Instead, only the altitude and azimuth angles of the Sun with respect to the camera are determined, and the angles of the Sun with respect to each heliostat are assumed to be the same as with respect to the camera.

After entering the required GPS information, the user is prompted to input the estimated location of the target in order to improve the accuracy of tracking the tower. The depth to the tower is the only critical dimension, as it is used to calculate the horizontal and vertical offsets. If the user selected the option to track a light source, the program will prompt the user to input the estimated location of the light source after inputting the location of the tower. As with the tower, the depth to the light source is the only critical dimension.

- **State 0:** State 0 is an initialization state which welcomes the user and prompts the user to select an option: 1 to track the Sun, and 2 to track a light source.

- State 1:** State 1 waits for an input from the user. If the user types a 1, the sun will be tracked. Upon transitioning to State 2, the flag `real_sun` is set high, and `option_selected` is set to 1. If the user types a 2, a light source will be tracked. Upon transition to State 2, `option_selected` is set to 2. If the user selected to track the sun, the task will advance to State 2 and prompt the user to select whether to use GPS coordinates to estimate the position of the Sun, or to scan for the Sun instead.
- State 2:** State 2 waits for the user to select to track the Sun using GPS coordinates or to scan instead. If the option to use GPS is selected, the user is prompted to input the longitudinal location of the camera, and the task proceeds to State 3 after setting the `use_gps_flag` high and setting `gps_num` to 1. If the option to use GPS is not selected, the user is prompted to input the depth of the tower with respect to the camera, and `tower_num` is set to 1 upon transition to State 3.
- State 3:** State 3 calculates the input from a user. The user is allowed to enter up to five number characters and a negative sign. The program calculates the input by storing the inputted values into a buffer as they are typed in. Each time the user inputs a number, the variable `data_count` is incremented to keep track of the number of characters entered. If backspace is hit, `data_count` is decremented if `data_count` is greater than 0 and the pointer to the buffer is also decremented. When the user hits enter, the first value stored in the buffer is multiplied by 10^n , where n is the number of characters entered. The next value stored is multiplied by 10^{n-1} and added to the previous value. This continues until the last number in the buffer is reached. If the negative sign is entered, `neg_flag` gets set high, and

the calculated value will be multiplied by -1 before being stored into the proper value. However, some values would not make sense to be negative, such as the month, hours, days, etc. In these cases, the negative flag is disregarded when the calculated value is stored into the appropriate variable.

- **States 4-11:** States 4-11 are responsible for storing the calculated values from State 3 in the appropriate GPS variables. Upon exiting the States 4-10, `gps_num` is incremented. Upon exiting State 11, `gps_num` is set back to zero and `tower_num` is set to 1.

Table 1: GPS Variable States

State	Input
4	Longitude
5	Latitude
6	Angle offset from North
7	Month
8	Day
9	Year
10	Hour of the day
11	Minutes of the hour

- **States 12-14:** States 12-14 are responsible for variables associated with the tower location. The State 12 stores the calculated value into `tower_1_y`, the depth of the tower with respect to the camera. States 13 and 14 store the values into `tower_1_x` and `tower_1_z`, respectively. Upon exiting States 12 and 13, the variable `tower_num` is incremented. Upon exiting State 14, `tower_num` is set to zero and `sun_num` is set to 1 if `option_flag` is equal to 2, meaning the user selected to track the light source. If `option_flag` is 1, all necessary inputs from the user have been recorded and the task transitions to State 18.

- **States 15-17:** States 15-17 are responsible for values associated with the location of the light source. State 15 stores the calculated value into `sun_1_y`, the depth of the light source with respect to the camera. States 16 and 17 store the values into `sun_1_x` and `sun_1_z`, respectively. Upon exiting States 15 and 16, `sun_num` is incremented and the task transitions back to State 3. The task always transitions from State 17 to State 18.
- **State 18:** State 18 is a dead state that sets `UI_done_flag = 1`, which signals that all necessary inputs have been recorded.

Figure 23: User Interface State Transition Diagram

Move Mirror X Task

Each heliostat has its own Move Mirror task. The functions of this group of tasks are to calculate the desired azimuth and altitude angles based on the vertical and horizontal offsets explained above, and move the mirrors to those desired orientations.

The desired orientations are calculated only after the tower and Sun or light source locations are determined by the Move Camera task.

- **State 0:** State 0 waits for the Move Camera task to complete calculating the global positions of the tower and light source (if the option is selected), at which time it will set the move_mirror_X flag high. The task will then transition to State 1.
- **State 1:** State 1 is the calculation state, where the global positions of the tower and light source, or angles of the Sun, are used to determine the desired azimuth and altitude angles of the heliostat. Once the calculations are complete, the task transitions to State 2.
- **State 2:** State 2 moves the heliostat's servos to the desired orientations calculated in State 1. When the desired orientations are reached, the task transitions back to State 0 and sets the move_mirror_X flag low.

Figure 24: Move Mirror X State Transition Diagram

Blob Detection Function

The blob detection used in this project was included with the firmware for the SRV-1 Blackfin Camera and is distributed under the GNU General Public License.

In general, a blob detection function groups up pixels in an image into blobs based on colors. In this program, the image is scanned for pixels that have a color definition that falls in the range of colors specified by a particular bin. The camera outputs images in YUV color space, where Y is the brightness of the pixel, and U and V are chrominance, or color, components. For detecting the tower inside, the color bin has the following ranges: $50 \leq Y \leq 150$, $50 \leq U \leq 125$, $150 \leq V \leq 255$. For detecting the

tower outside, the color ranges have to be adjusted based on the amount of light present. For detecting the Sun or light source, the color ranges are: $254 \leq Y \leq 255$, $123 \leq U \leq 132$, $123 \leq V \leq 132$. Pixels that are found within the color ranges that are in close proximity to one another are grouped up into blobs. Blobs that fall under a minimum size are discarded, while the rest are stored into an array ordered by size, with the largest blob stored at the first location. The result of the blob detection algorithm can be seen below in Figures 25 through 28. Note: blobs are not centered.

Figure 25: Outside tower source image

Figure 26: Outside tower with blob detection enabled

Figure 27: Sun source image

Figure 28: Sun image with blob detection enabled

Move Servo Function

The move servo function handles the movement of each servo. The position of a servo is controlled by sending a pulse every 20 milliseconds. These servos do not have speed control, so they will move to the position specified as quickly as possible. The function smoothes out the motions of the servos by adding incremental points between the current and desired locations. The function takes in parameters of servo number and desired position. The desired position is compared to the current position of the servo, and if the desired position is different than the current position, the servo moves one step, corresponding to a change in pulse width of 0.5 microseconds, toward the desired position. Once the servo moves one step, the current position is updated.

When the current position of the servo is equal to the desired position of the servo, a flag is set to signal that the servo has reached the desired position.

Detect Color Function

The detect color function determines the center points of the largest blob in the camera's image corresponding to a particular color. The function sets the color bins based on the color to track, and calls the blob detection function described above. It calculates the center of the largest blob found by averaging the maximum and minimum pixel coordinates of the blob in the vertical and horizontal directions. The function returns a 1 if a blob has been detected and a 0 if no blobs have been detected.

Track Color Function

This function tracks a desired color specified by the color bins set in the detect color function by moving the camera servos to center the middle of the blob in the center of the camera's view. The location of the center of the blob in the camera's view is determined by the detect color function described above. If the center of blob lies outside of the range of values corresponding to the center of the camera's image, then the servos of the camera are commanded to move one step toward the blob center.

There are two different ranges used for aligning the blob's center with the camera's center—a large range of ± 20 pixels, and a slim range of ± 1 pixel. The camera scans at full speed until the large range is found, proceeding with one motion for every frame captured. When the blob is found within the large range, three images are captured and the blob centers are averaged between each movement of the camera. This slows down the movement speed of the camera, preventing overshoot due to long processing times, while smoothing out the tracking.

The processing time of the image is directly related to the capture resolution. A capture resolution of 160x120 yields very little delay between capture and processing while offering less information, and a resolution of 640x480 yields long delays, but offers more information. A 320x240 capture resolution is used in this program as a compromise.

2.1.3 Transformation Relationships

The location of the target with respect to the camera axes of rotation (universe frame) is determined based on the altitude and azimuth angles of the camera when the target is centered via blob tracking. Figure 29 shows the frames attached to each element of importance.

Figure 29: Coordinate references

To calculate the horizontal offset of the target with respect to the universe frame (x_t), the depth of the target (y_t) is multiplied by the tangent of the azimuth angle of the camera:

$$x_t = y_t * \tan (az)$$

The vertical offset of the target with respect to the universal frame (z_t) is determined by multiplying the magnitude of the vertical and horizontal offsets by the tangent of the altitude angle of the camera:

$$z_t = \sqrt{y_t^2 + x_t^2} * \tan(alt)$$

The position of the tower and Sun with respect to each heliostat is calculated using transformation matrices, as presented in *Introduction to Robotics: Analysis, Systems, Applications* [24]. The universe frame, T_U , is specified as:

$$T_U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The location of the tower with respect a mirror ${}^M T_t$ is calculated as follows:

$${}^M T_t = {}^M T_U * {}^U T_t$$

where ${}^M T_U$ is the transformation of the universe frame with respect to the mirror, and is found by inverting the transformation of the mirror with respect to the universe frame, ${}^U T_M$. ${}^U T_M$ is found by translating x_M , y_M , and z_M , along the X, Y, and Z axes, respectively, followed by a rotation of θ_1 about the a_M axis and a rotation of θ_2 about the n_M axis:

$${}^U T_M = \text{trans}(x_M, y_M, z_M) * T_U * \text{rot}(a_M, \theta_1) * \text{rot}(n_M, \theta_2)$$

$${}^M T_U = ({}^U T_M)^{-1} = [\text{trans}(x_M, y_M, z_M) * T_U * \text{rot}(a_M, \theta_1) * \text{rot}(n_M, \theta_2)]^{-1}$$

Thus,

$${}^M T_t = [\text{trans}(x_M, y_M, z_M) * T_U * \text{rot}(a_M, \theta_1) * \text{rot}(n_M, \theta_2)]^{-1} * \text{trans}(x_t, y_t, z_t) * T_U$$

Substituting in the transformation matrices yields:

$${}^M T_t = \left(\begin{bmatrix} 1 & 0 & 0 & x_M \\ 0 & 1 & 0 & y_M \\ 0 & 0 & 1 & z_M \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C\theta_1 & -S\theta_1 & 0 & 0 \\ S\theta_1 & C\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & C\theta_2 & -S\theta_2 & 0 \\ 0 & S\theta_2 & C\theta_2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Simplifying the above equation yields the transformation of the location of the target with respect to the mirror as:

$${}^M T_t = \begin{bmatrix} C\theta_1 & S\theta_1 & 0 & C\theta_1(x_t - x_M) + S\theta_1(y_t - y_M) \\ -S\theta_1 C\theta_2 & C\theta_1 C\theta_2 & S\theta_2 & S\theta_1 C\theta_2(x_t - x_M) + C\theta_1 C\theta_2(y_t - y_M) + S\theta_2(z_t - z_M) \\ S\theta_1 S\theta_2 & -C\theta_1 S\theta_2 & C\theta_2 & S\theta_1 S\theta_2(x_t - x_M) + C\theta_1 S\theta_2(y_t - y_M) + C\theta_2(z_t - z_M) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The angles θ_1 and θ_2 (azimuth and altitude angles) and distance along the mirror normal to the center of the target (P_y) can be calculated by solving the following three equations:

$$C\theta_1(x_t - x_M) + S\theta_1(y_t - y_M) = 0$$

$$S\theta_1 C\theta_2(x_t - x_M) + C\theta_1 C\theta_2(y_t - y_M) + S\theta_2(z_t - z_M) = P_y$$

$$S\theta_1 S\theta_2(x_t - x_M) + C\theta_1 S\theta_2(y_t - y_M) + C\theta_2(z_t - z_M) = 0$$

If only the position of the target is desired, θ_1 and θ_2 can be set to zero. This simplifies the above matrix to:

$${}^M T_t = \begin{bmatrix} 1 & 0 & 0 & x_t - x_M \\ 0 & 1 & 0 & y_t - y_M \\ 0 & 0 & 1 & z_t - z_M \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which yields:

$${}^M x_t = x_t - x_M$$

$${}^M y_t = y_t - y_M$$

$${}^M z_t = z_t - z_M$$

Thus, the position of the target with respect to the heliostat is the difference of the position of the target with respect to the universal frame and the location of the heliostat with respect to the universal frame. The location of the light source with respect to each heliostat can be calculated the same way. If the sun is being tracked, then the altitude and

azimuth angles of the sun with respect to the universal frame are directly transferred to the other heliostats, assuming that the sun is so far away that the difference in angle between heliostats is negligible.

In order to calculate the necessary orientation of a particular heliostat, the vectors of the target and light source with respect to the heliostat are bisected. This is done using the vector equation:

$$\vec{c} = |\vec{a}|\vec{b} + |\vec{b}|\vec{a}$$

Where c is the bisecting vector, and a and b are the vectors of the target and light source, respectively.

Figure 30: Bisecting vector representation

This can be proved algebraically:

$$\text{Let } \vec{c} = |\vec{a}|\vec{b} + |\vec{b}|\vec{a}$$

$$\cos \angle \vec{a}, \vec{c} = \frac{\vec{a} \cdot \vec{c}}{|\vec{a}||\vec{c}|} = \frac{\vec{a} \cdot (|\vec{a}|\vec{b} + |\vec{b}|\vec{a})}{|\vec{a}||\vec{c}|} = \frac{|\vec{a}|(\vec{a} \cdot \vec{b}) + |\vec{b}||\vec{a}|^2}{|\vec{a}||\vec{c}|} = \frac{(\vec{a} \cdot \vec{b}) + |\vec{b}||\vec{a}|}{|\vec{c}|}$$

$$\cos \angle \vec{c}, \vec{b} = \frac{\vec{b} \cdot \vec{c}}{|\vec{b}||\vec{c}|} = \frac{\vec{b} \cdot (|\vec{a}|\vec{b} + |\vec{b}|\vec{a})}{|\vec{b}||\vec{c}|} = \frac{|\vec{b}|(\vec{a} \cdot \vec{b}) + |\vec{b}||\vec{a}|^2}{|\vec{b}||\vec{c}|} = \frac{(\vec{a} \cdot \vec{b}) + |\vec{b}||\vec{a}|}{|\vec{c}|}$$

$$\cos \angle \vec{a}, \vec{c} = \cos \angle \vec{c}, \vec{b}$$

If the sun is being tracked, a vector is created based on the known altitude and azimuth angles of the sun, in the following manner:

$$\vec{b} = \sin(az_s) \hat{i} + \cos(az_s) \hat{j} + \tan(alt_s) \hat{k}$$

Chapter 3: Experiments and Results

The experiments presented in this chapter are meant to test the ability of the camera to pick up small movements, and correct the desired orientations of the surrounding heliostats accordingly. The emphasis in testing is on detecting and correcting for lateral movements, as heliostats are most often oriented with the receiver tower from north to south with prevailing winds from west to east. In addition to lateral movements, vertical movements are also tested.

3.1 Calibrating the Camera

The camera had two distinct areas that needed to be calibrated in order to extract valuable data. First, the camera had to be calibrated based on the brightness in the image, using white balance, gain correction, and exposure control. Second, the number of pixels per real-world distance had to be calculated.

3.1.2 White Balance, Gain Correction, and Exposure Control

White balance correction is very important when using blob detection algorithms that involve color thresholds. Although it is easy for humans to determine colors under different lighting conditions (i.e. a viewer can determine that a red ball is red whether in the sun or shade), it is more difficult for a computer to determine whether a ball is red based on the image taken from a camera, since it can only judge color based on digital values. The color values of the pixel are heavily dependent on the lighting conditions, and may fall out of range of values used for blob detection.

The OV7725 camera has built in auto white balancing, gain correction, and exposure control. The heliostat program prompts the user to place a white sheet of paper

in front of the camera upon startup in order to calibrate white balancing. This enables the camera to balance itself based on the ambient light and to capture images with the colors they seem in real life. The gain correction and exposure control adjust the brightness of the image to not saturate the pixels. Gain correction scales the brightness of pixels based on the maximum and minimum brightness of the pixels, and the exposure control adjusts the amount of light that enters the camera by changing the exposure time. If the exposure control is left on, then as soon as the light source comes into the frame everything will be dimmed in order to not over-saturate the pixels. The result of this is that the light source appears as a bright spot in the image, while everything else appears dark, and the pixels corresponding to the target fall out of range of the color thresholds specified. However, while tracking the Sun or light source the automatic exposure control is turned back on. If the automatic exposure control is left off, then the light source will over-saturate the image and the program will determine the light source to be in an incorrect position. If the sun comes within the same image as the target, it could over-saturate the image unless the target is near equal brightness. If the target is too dark, a system using multiple cameras can be put in place. For example, a camera on the other side of the tower will not see the sun in its image when looking at the tower, and can measure the sway in the tower and communicate the information to the heliostats on the other side.

For this project, the white balance, gain correction, and exposure control settings are determined through the program's operation.

3.1.3 Determining the Pixel to Distance Relationship

In order to determine the minimum angular offset that can be detected with this camera, a pixel-to-distance relationship had to be determined. The camera has three

usable image capture resolutions sizes: 640x480, 320x240, and 160x120. For calibration, the image capture resolution was set to 640x480.

The camera was calibrated by placing a 4.5-inch by 4.5-inch 9x9 checkerboard pattern with 0.5-inch squares 3 inches in front of the camera.

Figure 31: Setup of camera calibration. The square checkerboard pattern is placed at a depth of 3 inches from the camera.

The picture from the camera's perspective can be seen in Figure 32 at a resolution of 640x480 pixels, with the pixel lengths of several squares marked. The distortion can be observed visually, as well as by noting the different widths of the squares in terms of pixels. An 18-inch by 18-inch checkerboard with 2-inch squares was also used to calibrate the camera, at a depth of 12 inches, and the results were the same, showing that the distortion of the image is constant at different depths.

Figure 32: Camera's view of the checkerboard in Figure 27 at 640x480 resolution, with pixel lengths specified for several squares. Each square is 0.5 inches by 0.5 inches.

When these pixel lengths are plotted against the distance from the center of the image in inches, a non-linear relationship can be seen, described by Figure 33.

Figure 33: Distortion due to camera lens. The further away from the center the pixel is, the more deviant it is from actual position.

Figure 34: Linear portion of camera lens distortion with up to 14 degrees of linearity.

With a 640x480 capture resolution, the center square spans a total of 96 pixels.

The angle from the center of the camera to the edge of the square is calculated below:

$$\alpha = \text{atan}\left(\frac{0.25''}{L}\right)$$

With $L = 3$ inches, α is 4.76° . The pixel distribution with regards to global position is assumed to be linear up to 14 degrees offset from the center of the image (see Figure 34).

Under this assumption, the number of pixels per degree from the center of the image is found to be:

$$\frac{\#pixels}{degree} = 0.0158 * i_w$$

where i_w is the total width of the image. At different resolutions, the number of pixels for one degree offset can be seen below in Table 2. Inverting these numbers gives the minimum angle offset that can be detected with the particular resolution. For example, with an image width of 160 pixels, the minimum angle offset that can be detected is 0.4

degrees from center. Likewise, this equates to an angle offset of 0.2 degrees for an image width of 320 pixels, and 0.1 degrees with an image width of 640 pixels.

Table 2: Number of pixels per degree offset

i_w (pixels)	# of pixels per degree offset
160	2.5
320	5.0
640	10.1

3.2 Calibrating the Servos

Each servo was calibrated by attaching a protractor to its output shaft and sending digital signals to the servo and checking the corresponding angle (see Figure 37). The angular position of the shaft was recorded for digital values of 0, 125, 250, 375, and so on up to 4000. The calibration data for servos 1 and 2, corresponding to the azimuth and altitude of the camera, respectively, is plotted in Figures 35 and 38. Plots for the remaining servos are attached in the appendix.

Figure 35: Camera Azimuth Servo Calibration (Servo 1)

Figure 36: Camera Altitude Servo Calibration (Servo 2)

Figure 37: Servo Calibration Setup. A protractor is attached to the output shaft of the servo.

3.3 Determining the Position of the Target

Figure 38: Target positioned at a depth of 24 inches and 0 inches horizontal offset. Each square is 6x6 inches.

In order to test the ability to detect movement in the target, the target was initially placed at a depth of 24 inches from the camera, a vertical offset of 24 inches, and no horizontal offset. The program was turned on and ran until the target and light source were found. Then, the program moved back to the target and found it again. Once the target was found the second time, the console printed out the determined location of the target, as well as the desired altitude and azimuth angles for each heliostat. Next, the target was shifted left 0.40 inches corresponding to a one-degree shift in the tower with respect to the camera, and the camera scanned again to determine the new location of the target. The new location and desired orientations of each mirror were then printed out in the console. This process was repeated for target shifts of 0.4 inches right, 6 inches left, 6 inches right, and 0.4 inches down. Then, the target was moved to depths of 36 and 48 inches, shifted left and right 1 degree, shifted left and right 6 inches, and shifted down 1

degree. At a depth of 48 inches, the target was also shifted left and right half a degree. The angular errors can be seen in Figure 39. The average error was found to be 0.32 degrees, with minimum error of 0.01 degrees and maximum error of 1.08 degrees.

Figure 39: Average angular offset error

Plots comparing the determined azimuth angles and actual azimuth angles with respect to the camera can be seen in Figures 40 through 43.

Figure 40: Azimuth angular displacements at a depth of 24 inches

Figure 41: Azimuth angular displacements at a depth of 36 inches

Figure 42: Azimuth angular displacements at a depth of 48 inches

Figure 43: Average Vertical displacement of the tower with respect to the camera

When the target was displaced vertically by 0.5 degrees with respect to the camera, the program correctly recognized the amount of displacement. However, the program incorrectly determined 0.75 and 1 degree shifts in the target by 0.6 and 0.32 degrees, respectively.

3.4 Accuracy of Prototype

In this test, the centers of the reflected images of the light source are compared with the center of the target for both indoor and outdoor testing. The mirrors are referred to as 1-3 according to Figure 44.

Figure 44: Mirror Number Designation

Outdoor

The target was placed at a depth of 24 inches from the camera, no horizontal offset, and 24 inches high. The reflected images were recorded at three different times of the day: 8:45am, 9:05am, and 9:35am. The offset of each reflected image from the center of the target was recorded and used to calculate the angular error of each respective heliostat.

Figure 45: Reflected Solar Images at 8:45am.

Figure 46: Reflected Solar Images at 9:05am

Figure 47: Reflected Solar Images at 9:35am

Figure 48: Radial offsets of the reflected solar images at different times of the day

In these trials, the angular errors of the mirrors range from 0.4 degrees up to 2.6 degrees.

Indoor

For this experiment, the light source was located at a depth of 24 inches, a horizontal offset of -36 inches, and a height of 78 inches with respect to the camera. The target was located at a depth of 24 inches, horizontal offset of -6, -1, 0, +1, and +6 inches, and a height of 24 inches with respect to the camera. The centroids of the reflected light images were recorded and used to calculate the angular errors of each respective heliostat (see Figure 49).

Figure 49: Radial offsets of reflected images on target

The errors recorded in this trial ranged from 0.1 degrees to 1.2 degrees.

3.5 Depth Study

In order to determine the effects of tower sway in a front-to-back motion (thereby changing the depth from the camera to the target), the errors in the angle bisector are calculated assuming the target is at a depth from the camera equal to its height.

Figure 50: Angular errors associated with depth offset

For example, the target would be at a depth of 24 inches and a height of 24 inches, with no horizontal offset. The depth offset is calculated as a percentage of the total depth from the camera to the target. A negative depth means the target has moved toward the camera, and a positive depth means the target has moved away from the camera. As the change in depth increases, the angular error also increases. A 0.5% offset yields angular errors below 0.1 degrees for each surrounding heliostat.

3.6 Correcting for Movements of the Base

In this experiment, the model is tilted by 3.5 degrees left, right, forward, and backward. The target is located at a depth of 24 inches, with horizontal offsets of -6, 0, and +6 inches, and height of 24 inches. The resulting errors of the mirror normals can be seen below in Figures 53 through 55. The model was tilted by placing two 1x4 planks underneath a $\frac{3}{4}$ " piece of plywood (see Figures 51 and 52).

Figure 51: Base tilted left

Figure 52: Base tilted forward

Figure 53: Mirror 1 angular error after correcting for base tilt

Figure 54: Mirror 2 angular error after correcting for base tilt

Figure 55: Mirror 3 angular error after correcting for base tilt

The model was able to correct for tilting of the base for mirror 1 to within 1.6 degrees. Mirror 2 was corrected to within 5.5 degrees, and mirror 3 was corrected to within 1.2 degrees when the base was tilted left, right, and forward. The mirrors had the most errors when the base was tilted backward.

Chapter 4: Conclusions

4.1 Conclusions

In this thesis, a prototype of a method for tracking the sway in central receiver towers was constructed. The prototype consisted of a CMOS camera connected to a microcontroller with three surrounding heliostats. The model was able to detect movements in the tower with average error of 0.32 degrees, and was able to correctly orient the heliostats to within 1.2 and 2.6 degrees of accuracy while testing indoors and outdoors, respectively.

With tower sway estimated to be a maximum of 0.5% of the tower height, this model would not suffice in tracking such small motions. However, it does serve as a stepping stone to a more advanced system. With a higher resolution camera and higher resolution actuators, it may be possible to detect the small motions associated with tower sway.

4.2 Future Work

In order to progress this method further, different cameras and drive mechanisms should be looked into. It may make more sense to use higher-resolution cameras connected to more powerful central computers in a concentrated solar power plant. Higher resolution would increase the accuracy of detecting any sway in the tower, as presented in Section 3.1.3. More work also needs to be done to understand the nature of the tower sway, and determine whether it is worth correcting for. If the tower sways to the point where heliostats are off target, it may make more sense to just increase the width of the tower to account for the sway. Since the sway is periodic, constantly

running the heliostat actuators would end up costing a significant amount of energy—likely more than the energy gain from correcting for offsets.

The vision algorithms could certainly be improved upon as well. Calibration methods based on the amount of light present could greatly improve the accuracy and reliability of detecting both the tower and the Sun. With the amount of light changing throughout the day, as well as the amount of light falling on the target, it is difficult to reliably detect the tower based on simple color thresholding.

Clouds proved to be a significant source of background noise, as they reflect sunlight near the intensity of the sun. As a result, the blob tracking algorithms incorrectly interpreted clouds as instances of the sun, and determined the sun to be in a different location than it actually was. In order to correct for this, more sophisticated algorithms can be utilized such as looking specifically for circles and a more adaptive light correction scheme. Tightening the tolerances of the color detection or using lens filters to decrease the amount of light entering the camera could also improve the results.

References

- [1] eSolar Company. www.esolar.com
- [2] Wilson TurboPower. www.wilsonsrbopower.com
- [3] Kribus, A. *et al.* "Closed loop control of heliostats." *Energy*. 2004.
- [4] "Guidelines for Performance-Based Seismic Design of Tall Buildings November 2010." Pacific Earthquake Engineering Research Center (PEER). pp 51-52.
- [5] Blitz, D. "Solar Energy Sun Tracker." *U.S. Patent No. 4,013,885*. 1977.
- [6] Vaerewyck, E. *et al.* "Heliostat Guidance." *U.S. Patent No. 4,211,922*. 1978.
- [7] Berenguel, M., Rubio, F.R., Valverde, A., *et al.* "An artificial vision-based control system for automatic heliostat positioning offset correction in a central receiver solar power plant." *Solar Energy*. 2004.
- [8] power-technology.com. www.power-technology.com/projects/Seville-Solar-Tower/
- [9] SolarPACES.org. <http://www.solarpaces-csp.org/Tasks/Task1/ps10.htm>
- [10] Dumiak, M. "Simple and bright, heliostats tap sunlight for lighting outdoor and, increasingly, indoor spaces." *Architectural Record*.
<http://archrecord.construction.com/tech/techbriefs/0705dignews-1.asp>
- [11] Inhabitat.com. <http://inhabitat.com/cold-light-fiberoptic-lamp-shines-sunlight-indoors/>
- [12] Practical Solar. www.practicalsolar.com
- [13] Rohr, B. "The Promise of Small Heliostats." *Northeast Sun*. Spring 2009.
- [14] Solare-Bruecke.com. "The Scheffler-Reflector". http://www.solare-bruecke.org/English/scheffler_e-Dateien/scheffler_e.htm

- [15] Pasachoff, J., Livingston, W. "Coelostat and heliostat: alignment and use for eclipse and other field purposes." *Applied Optics*. Vol. 23, No.16. Aug. 1984.
- [16] Altmeyer, S., Frank, J. "Optics and Information Technology." Springer Berlin Heidelberg. 2009. pp. 98-99.
- [17] Litwiller, D. "CCD vs. CMOS: Facts and Fiction". *Photonics Spectra*, Jan. 2001.
- [18] Ridgely, J. "Using Tasks and Finite State Machines in C++".
- [19] Reznik, D. "Camera-based heliostat tracking controller." *U.S. Patent No. 12/497,385*. Jan. 2011.
- [20] Heliotrack.com. <http://www.heliotrack.com/Heliostat1M.html>
- [21] Yogev, A. *et al.* "Control of a Heliostat Field in a Solar Energy Plant." *U.S. Patent No. 5,862,799*. 1997.
- [22] Ulmer, S. *et al.* "Automated high resolution measurement of heliostat slope errors." *Solar Energy*. Jan. 2010.
- [23] Surveyor Corporation. www.surveyor.com
- [24] Niku, S. "Introduction to Robotics: Analysis, Systems, Applications." Prentice Hall. 2001.

Appendix

Appendix A: Programming the Blackfin

To program the SRV-1 Blackfin, first download the most recent firmware from www.surveyor.com. Replace the myfunc.c and myfunc.h files with the files attached. Connect the computer to the Blackfin via the USB-to-UART serial converter. Download a terminal program such as TeraTerm. Select the serial connection option (a COM port should be listed). Set up the connection for 115,200 baud rate, 8 bits, No parity, and 1 stop bit. Once this is set up, type a 'V' to get the version of the firmware loaded. To upload a new program to the Blackfin, type 'X'. There should be C's printing out in the console. Perform an XMODEM transfer, and send the new srv.ldr file. After the file transfer is complete, type 'zZ' to load the program into memory. Type '\$!' or push the reset button the reset the processor.

To compile files, download a MAKE compiler. Open the command prompt (type 'cmd' into the search area in Windows Vista or Windows 7) and navigate to the file where the firmware is located. Type "make clean all" to compile the firmware and create a new srv.ldr file that can be uploaded to the Blackfin.

Appendix B: Servo Calibration Plots

Azimuth Mirror 1

Altitude Mirror 1

Azimuth Mirror 2

Altitude Mirror 2

Azimuth Mirror 3

Altitude Mirror 3

Appendix C: Program Code

```
// myfunc.h
// Description: This header file contains the defines, variables, and
//             function declarations for the tower-tracking heliostat program.
// Created:    3/9/11 by Joel Masters

// Defines
#define WHITE 1           // white for blob detection
#define BLACK 2          // black for blob detection
#define RED 3            // red for blob detection
#define MAX_ALTITUDE 3000 // mazimum altitude digital value
#define AZ1 0            // servo number azimuth for camera
#define ALT1 1           // servo number altitude for camera
#define AZ2 2            // servo number azimuth for mirror 2
#define ALT2 3           // servo number altitude for mirror 2
#define AZ3 4            // servo number azimuth for mirror 3
#define ALT3 5           // servo number altitude for mirror 3
#define AZ4 6            // servo number azimuth for mirror 4
#define ALT4 7           // servo number altitude for mirror 4
#define MAX_ALT 3000     // max altitude, due to mechanical constraints digital
value
#define MAX_MM_COUNT 50000 // count to cycle through mastermind before transition
#define R2D 57.29578      // radians to degree conversion factor
#define D2R 0.017453      // degrees to radians conversion factor
#define MAX_AVE_NUM 3      // number of frames used for averaging
#define MAX_FRAME_COUNT 3 // number of frames captured before moving in close
proximity
#define WIDE_RANGE 20      // pixel range of error for fast scanning
#define SLIM_RANGE 2       // pixel range of error for fine scanning
#define IMAGE_WIDTH 320    // image width
#define IMAGE_HEIGHT 240   // image height
#define MIN_BLOB_WIDTH 10  // minimum blob width
#define MIN_BLOB_HEIGHT 10 // minimum blob height
#define MAX_BLOB_WIDTH 80  // maximum blob width
#define MAX_BLOB_HEIGHT 80 // maximum blob height
#define IMAGE_PIX_RATIO 0.198 // for 320x240 resolution, 5.1 pixels per degree
#define BORDER 10          // restricted area around frame blobs cannot fall into
#define DELAY_TIME 5       // ms for time delay between servo movements

// GPS defines
// declaration of months and their respective number of days
#define jan 31 // number of days in january
#define feb 28 // number of days in february
#define mar 31 // number of days in march
#define apr 30 // number of days in april
#define may 31 // number of days in may
#define jun 30 // number of days in june
#define jul 31 // number of days in july
#define aug 31 // number of days in august
#define sep 30 // number of days in september
#define octo 31 // number of days in october
#define nov 30 // number of days in november

// other constants
#define timezone 8 // PST

// Variables
char image_capture_state; // image capture task state
char cap_flag;           // flag to capture images
char move;               // move flag for testing
char img_process;        // flag to signal the image has been processed
char mov_camera;         // flag to move mirror 1
char mov_mirror_2;       // flag to move mirror 2
char mov_mirror_3;       // flag to move mirror 3
char mov_mirror_4;       // flag to move mirror 4
char servo_zero;         // flag to move to zero position
char az1flag;            // flag for az 1 position
char alt1flag;           // flag for alt 1 position
```

```

char az2flag;           // flag for az 2 position
char alt2flag;          // flag for alt 2 position
char az3flag;           // flag for az 3 position
char alt3flag;          // flag for alt 3 position
char az4flag;           // flag for az 4 position
char alt4flag;          // flag for alt 4 position

float mirror_2_x;        // x position of mirror 2 relative to camera
float mirror_2_y;        // y position of mirror 2 relative to camera
float mirror_2_z;        // z position of mirror 2 relative to camera

float mirror_3_x;        // x position of mirror 3 relative to camera
float mirror_3_y;        // y position of mirror 3 relative to camera
float mirror_3_z;        // z position of mirror 3 relative to camera

float mirror_4_x;        // x position of mirror 4 relative to camera
float mirror_4_y;        // y position of mirror 4 relative to camera
float mirror_4_z;        // z position of mirror 4 relative to camera

char mov_camera_state;   // move camera state
unsigned int az_camera;   // azimuth of the camera
unsigned int desired_az_camera; // desired azimuth of camera
unsigned int alt_camera;  // alt of camera
unsigned int desired_alt_camera; // desired alt of camera

char mov_mirror_2_state; // move mirror 2 state
unsigned int az_mirror_2; // azimuth of mirror 2
unsigned int desired_az_mirror_2; // desired azimuth of mirror 2
unsigned int alt_mirror_2; // altitude of mirror 2
unsigned int desired_alt_mirror_2; // desired altitude of mirror 2

char mov_mirror_3_state; // move mirror 3 state
unsigned int az_mirror_3; // azimuth of mirror 3
unsigned int desired_az_mirror_3; // desired azimuth of mirror 3
unsigned int alt_mirror_3; // altitude of mirror 3
unsigned int desired_alt_mirror_3; // desired altitude of mirror 3

char mov_mirror_4_state; // move mirror 4 state
unsigned int az_mirror_4; // azimuth of mirror 4
unsigned int desired_az_mirror_4; // desired azimuth of mirror 4
unsigned int alt_mirror_4; // altitude of mirror 4
unsigned int desired_alt_mirror_4; // desired altitude of mirror 4

// GPS values
float gps_long;          // longitude
float gps_lat;           // latitude
float gps_angle;         // angle CW from north
unsigned short gps_month; // month
unsigned short gps_day;   // day
unsigned short gps_year;  // year
unsigned short gps_hours; // hour of the day
unsigned short gps_minutes; // minute of the hour
unsigned char gps_num;    // state number for GPS inputs in UI
unsigned char tower_num;  // state number for tower position inputs in UI
unsigned char sun_num;    // state number for sun position inputs in UI
unsigned char use_gps_flag; // flag to use GPS coordinates

// Zeros
float AZ_ZERO_1;         // azimuth 1 zero position
float ALT_ZERO_1;        // altitude 1 zero position
float AZ_ZERO_2;         // azimuth 2 zero position
float ALT_ZERO_2;        // altitude 2 zero position
float AZ_ZERO_3;         // azimuth 3 zero position
float ALT_ZERO_3;        // altitude 3 zero position
float AZ_ZERO_4;         // azimuth 4 zero position
float ALT_ZERO_4;        // altitude 4 zero position

// General temp variables
float f_temp;            // temp variable

```

```

float f_temp1;           // temp variable
float f_temp2;           // temp variable
float f_temp3;           // temp variable
float f_temp4;           // temp variable
float f_temp5;           // temp variable
float f_temp6;           // temp variable
float f_temp_az_bisect;  // temp variable for azimuth bisector
float f_temp_alt_bisect; // temp variable for altitude bisector

// Dimensions
int L;                   // depth of tower with respect to camera
int L_sun;               // depth of light source wrt camera

// Image Processing
int target_centerx_red;  // x center of red blob
int target_centery_red;  // y center of red blob
int target_centerx_white; // x center of white blob
int target_centery_white; // y center of white blob
int target_centerx_black; // x center of black blob
int target_centery_black; // y center of black blob
char centeredx;          // flag for x centered
char centery;            // flag for y centered
int img_bufx_red[MAX_AVE_NUM]; // array for x centers of red blob
int img_bufy_red[MAX_AVE_NUM]; // array for y centers of red blob
int* img_ptrx_red;        // pointer for x center array for red blobs
int* img_ptry_red;        // pointer for y center array for red blobs
int img_bufx_white[MAX_AVE_NUM]; // array for x centers of white blobs
int img_bufy_white[MAX_AVE_NUM]; // array for y centers of white blobs
int* img_ptrx_white;      // pointer for x center array for white blobs
int* img_ptry_white;      // pointer for y center array for white blobs
int img_bufx_black[MAX_AVE_NUM]; // pointer for x center array for black blobs
int img_bufy_black[MAX_AVE_NUM]; // pointer for y center array for black blobs
int* img_ptrx_black;      // pointer for x center array for black blobs
int* img_ptry_black;      // pointer for y center array for black blobs
char img_ptr_count_red;   // image pointer count for red blob array
char img_ptr_count_white; // image pointer count for white blob array
char img_ptr_count_black; // image pointer count for black blob array
char frame_count;        // counter for the number of frames taken
char wide_center_found;  // flag for wide center found
char slim_center_found;  // flag for slim center found
char start_flag_red;     // flag to start averaging red blob coordinates
char start_flag_white;   // flag to start averaging white blob coordinates
char start_flag_black;   // flag to start averaging black blob coordinates
char frame_count_flag;   // flag signalling enough frames have been captured
to move
float centeredx_offset;   // offset for centered x for more accuracy
float centery_offset;     // offset for centered y for more accuracy

// Program Flags
char real_sun;            // flag for tracking the real sun
char stop_flag;           // flag to stop and exit the program
char scan_up;             // flag to scan up (increase altitude)
char scan_left;          // flag to scan left
char sun_found;           // flag signalling the sun has been found
char tower_found;        // flag signalling the tower has been found
char temp_sun_found;      // flag signalling the sun has been found in the most
recent image
char temp_tower_found;    // flag signalling the tower has been found in the most
recent image
char find_sun;            // flag to tell the program to find the sun
char find_tower;          // flag to tell the program to find the tower
char sun_located;         // flag to signal the center of the sun has been found
char tower_located;       // flag to signal the center of the tower has been found
char track_tower;         // flag to tell the program to track the tower
char track_sun;           // flag to tell the program to track the sun
char move_to_tower;       // flag to move to known coordinates of the tower
char move_to_sun;         // flag to move to known coordinates of the sun
char sun_wide_center_found; // flag to signal the sun has been centered wide (to turn
AWB on)

```

```

char calc_pos;                // flag to signal that the coordinates of the sun and tower
                               // should be calculated
char data_flag;               // flag for UI
char done_flag;               // flag for UI
char calib_done;              // flag to exit calibration
char use_sun_pos_flag;        // flag for MM to move to sun location
char sun_found_count;         // number of times sun has been found
char first_time;              // first time through sun centered wide

// Processed values
float sun_az;                  // sun azimuth, digital value
float sun_alt;                 // sun altitude, digital value
float tower_az;                // tower azimuth, digital value
float tower_alt;               // tower altitude, digital value
float tower_1_x;               // tower x-coordinate wrt camera
float tower_1_y;               // tower y-coordinate wrt camera
float tower_1_z;               // tower z-coordinate wrt camera
float tower_2_x;               // tower x-coordinate wrt mirror 2
float tower_2_y;               // tower y-coordinate wrt mirror 2
float tower_2_z;               // tower z-coordinate wrt mirror 2
float tower_3_x;               // tower x-coordinate wrt mirror 3
float tower_3_y;               // tower y-coordinate wrt mirror 3
float tower_3_z;               // tower z-coordinate wrt mirror 3
float tower_4_x;               // tower x-coordinate wrt mirror 4
float tower_4_y;               // tower y-coordinate wrt mirror 4
float tower_4_z;               // tower z-coordinate wrt mirror 4
float sun_1_x;                 // sun x-coordinate wrt camera
float sun_1_y;                 // sun y-coordinate wrt camera
float sun_1_z;                 // sun z-coordinate wrt camera
float sun_2_x;                 // sun x-coordinate wrt mirror 2
float sun_2_y;                 // sun y-coordinate wrt mirror 2
float sun_2_z;                 // sun z-coordinate wrt mirror 2
float sun_3_x;                 // sun x-coordinate wrt mirror 3
float sun_3_y;                 // sun y-coordinate wrt mirror 3
float sun_3_z;                 // sun z-coordinate wrt mirror 3
float sun_4_x;                 // sun x-coordinate wrt mirror 4
float sun_4_y;                 // sun y-coordinate wrt mirror 4
float sun_4_z;                 // sun z-coordinate wrt mirror 4
float tower_2_az;              // tower azimuth wrt mirror 2, angle
float tower_2_alt;             // tower altitude wrt mirror 2, angle
float tower_3_az;              // tower azimuth wrt mirror 3, angle
float tower_3_alt;             // tower altitude wrt mirror 3, angle
float tower_4_az;              // tower azimuth wrt mirror 4, angle
float tower_4_alt;             // tower altitude wrt mirror 4, angle
float sun_2_az;                // sun azimuth wrt mirror 2, angle
float sun_2_alt;               // sun altitude wrt mirror 2, angle
float sun_3_az;                // sun azimuth wrt mirror 3, angle
float sun_3_alt;               // sun altitude wrt mirror 3, angle
float sun_4_az;                // sun azimuth wrt mirror 4, angle
float sun_4_alt;               // sun altitude wrt mirror 4, angle

// Camera Register Information
unsigned char data_store[0xC7]; // storage array for camera register values
unsigned char i2c_info[2];      // array to transfer i2c data
unsigned char data_buf[20];     // buffer for data for UI
unsigned char* buf_ptr;         // pointer to data buffer
unsigned char data_count;       // counter for pointer to data buffer

// Misc
int scan_up_count;              // number of counts to scan up
char mm_state;                  // state of mastermind task
unsigned int mm_count;          // number of times through mastermind
char move_mirror_num;           // servo number to move
char tower_watchdog;            // tower watchdog to prevent infinite loop
char sun_watchdog;              // sun watchdog to prevent infinite loop

// Track color variables
unsigned int step;               // digital value to move servo while scanning
#define SLIM_STEP 1              // slim step increment
#define WIDE_STEP 16             // wide step increment

```

```

// User Interface Variables
char UI_state; // state of user interface task
char option_flag; // option selected
char done_num_flag; // signalling number has been calculated
char neg_flag; // signalling a negative number
int UI_temp_calc; // calculated input value
char UI_done_flag; // signalling the UI is done

// Servo test variables
char pos_reached; // signalling the servo has moved to the correct
position
unsigned int current_pos; // current position of servo (corresponding to home
position)
unsigned char low_range; // low range proximity
unsigned char high_range; // high range proximity
#define start_byte 0x80 // start byte to send servo controller
#define device_ID 0x01 // device ID for servo controller
#define command 4 // command to send to servo controller
#define TEST_STEP 9 // number of steps to increment for testing

// Equation test variables
float test_temp; // test variable

int Lowx1; // low-end horizontal tolerance for blob centering
int Highx2; // high-end horizontal tolerance for blob centering

int Lowy1; // low-end vertical tolerance for blob centering
int Highy2; // high-end vertical tolerance for blob centering

// Function declaration of myfunc
void myfunc();

// Mastermind task
void task_mastermind(char);

// UI task
void task_UI(char);

// Function to capture an image
void task_image_capture(char);

// Function to move mirror 1
void task_mov_camera(char);

// Function to move mirror 2
void task_mov_mirror_2(char);

// Function to move mirror 3
void task_mov_mirror_3(char);

// Function to move mirror 4
void task_mov_mirror_4(char);

// Function to move the servos
void move_servo(unsigned char, unsigned int);

// Function to find a color in the image
// Returns a 1 if color found, 0 if color not found
char detect_color(int);

// Function to track a color
void track_color(char, char, char);

// Function to set the color bins
void set_color_bin(unsigned char);

// startup function for servos
void startup(void);

```

```

// Heliostat main function
void heliostat(void);

// Auto White balance turn on
void AWB_on(void);

// Auto White balance turn off
void AWB_off(void);

// Auto White Balance init
void AWB_init(void);

// servo test function
void servo_test(unsigned int, unsigned char);

// find altitude of the sun
float find_altitude (float, float, unsigned short, unsigned short, unsigned short,
unsigned short, unsigned short);

// find azimuth of the sun
float find_azimuth (float, float, unsigned short, unsigned short, unsigned short,
unsigned short, unsigned short, float);

// Servo calibration function
void calib(void);

// Sine
float f_sin(float);

// Cosine
float f_cos(float);

// Tangent
float f_tan(float);

// Arcsin
float f_asin(float);

// Arccos
float f_acos(float);

// Arctan
float f_atan(float);

// Atan2
float f_atan2(float, float);

// Square-root
float f_sqrt(float);

// Factorial
unsigned int f_factorial(unsigned int);

// Power
float f_power(float, int);

// Absolute value
float f_fabs(float);

// Calibration functions
float az_1_d2a(float);           // Azimuth for camera digital value to angle conversion
float az_1_a2d(float);           // Azimuth for camera angle to digital value conversion
float alt_1_d2a(float);           // Altitude for camera digital value to angle conversion
float alt_1_a2d(float);           // Altitude for camera angle value to digital conversion.

float az_2_d2a(float);
float az_2_a2d(float);
float alt_2_d2a(float);
float alt_2_a2d(float);

```

```

float az_3_d2a(float);
float az_3_a2d(float);
float alt_3_d2a(float);
float alt_3_a2d(float);

float az_4_d2a(float);
float az_4_a2d(float);
float alt_4_d2a(float);
float alt_4_a2d(float);

// Additional Calibration Functions
float az_calib(float);
float alt_calib(float);

// myfunc.c
// Description: This file contains tower-tracking heliostat program
//              heliostat() that uses the Surveyor SRV-1 Blackfin
//              Camera to determine the location of light source and
//              a colored target using blob tracking algorithms. The
//              program is started by typing '%t' into a terminal
//              program while connected to the Blackfin Camera.
// Created:     3/9/2011 by Joel Masters

#include "myfunc.h"
#include "print.h"
#include "gps.h"
#include "srv.h"
#include "i2c.h"
#include "colors.h"

void myfunc() {

    // initialize states
    mm_state = 0;
    UI_state = 0;
    image_capture_state = 0;
    mov_camera_state = 0;
    mov_mirror_2_state = 0;
    mov_mirror_3_state = 0;
    mov_mirror_4_state = 0;

    // initialize flags
    sun_found = 0;
    tower_found = 0;
    sun_located = 0;
    tower_located = 0;
    servo_zero = 0;
    stop_flag = 0;
    UI_done_flag = 0;

    // positions of mirrors wrt camera
    mirror_2_x = 9.5;
    mirror_2_y = 0;
    mirror_2_z = 0;
    mirror_3_x = 21;
    mirror_3_y = -6;
    mirror_3_z = 0;
    mirror_4_x = 16;
    mirror_4_y = 11;
    mirror_4_z = -1.75;

    // initialize positions of heliostats
    az_camera = 969;
    alt_camera = 969;
    az_mirror_2 = 969;
    alt_mirror_2 = 969;
    az_mirror_3 = 969;
    alt_mirror_3 = 969;
    az_mirror_4 = 969;

```



```

alt_mirror_4 = 969;

// for movement of servos
pos_reached = 0; // signalling the servo has moved to the correct position
current_pos = 1000; // current position of servo (corresponding to home
position)
low_range = 20; // low range proximity
high_range = 20; // high range proximity

// Pointers
buf_ptr = data_buf; // pointer to data buffer
img_ptrx_red = img_bufx_red; // pointer for x center array for red blobs
img_ptry_red = img_bufy_red; // pointer for y center array for red blobs
img_ptrx_white = img_bufx_white; // pointer for x center array for white
blobs
img_ptry_white = img_bufy_white; // pointer for y center array for white
blobs
img_ptrx_black = img_bufx_black; // pointer for x center array for black
blobs
img_ptry_black = img_bufy_black; // pointer for y center array for black
blobs

unsigned char ch;
ch = getch();

switch (ch) {
case 'f':
printf("AWB_off \r\n");
i2c_info[0] = 0x13;
i2c_info[1] = 0xC0 + (0 & 0x07); // turns AWB, AGC, AEC
off
i2cwrite(0x21, (unsigned char *)i2c_info, 1, SCCB_ON);
break;
case 'g':
printf("Camera Reset \r\n");
camera_reset(320);
delayMS(2000);
break;
case 'o':
printf("AWB_on \r\n");
AWB_on();
break;
case 'h': // test myfunc - string is "%h"
printf("myfunc: hello ! \r\n");
break;
case 'p':
// sets the servo controller to control servos 0-7
printf("Resetting Servo Numbers to 0-7. \r\n");
uart1SendChar(0x80);
uart1SendChar(0x02);
uart1SendChar(0);
break;
case 't':
// Run the heliostat program. Command to get here is '%t'
heliostat();
break;
case 'c':
// Calibration function
printf("Calibration. \r\n1 - Mirror 1 \r\n2 - Mirror 2\r\n3 -
Mirror 3\r\n4 - Mirror 4\r\n");
calib();
break;
default:
printf("Invalid Char\r\n");
break;
}

return;
}

```

```

// Image capture and processing function
// Function gets passed a char (image_capture_state) which specifies the state
// which the function is in. The function returns nothing.
void task_image_capture(char img_cap_state)
{
    char clr1_temp = 0;
    char clr2_temp = 0;
    int ix = 0;
    switch (img_cap_state)
    {
        case(0): // State 0: Initialization
            // turn auto-balance and correct off
            if (UI_done_flag)
            {
                camera_reset(320);
                printf("Adjusting light conditions... \r\n");
                delayMS(1000);
                printf("Hold up a white paper to the camera and hit any key.
\r\n");

                getch();
                printf("Calibrating... ");
                delayMS(1000);
                AWB_init();
                // Grab the camera register information
                for (ix = 0; ix < 0xC7; ix++)
                {
                    i2c_info[0] = ix;
                    i2c_read(0x21, (unsigned char *)i2c_info,1,SCCB_ON);
                    data_store[ix] = i2c_info[0];
                    printf("data_store[%d] = %d \r\n",ix,data_store[ix]);
                }

                printf("done. \r\n");
                printf("Remove white paper and hit any key to continue. \r\n");
                getch();
                printf("Running... \r\n");
                image_capture_state = 1;
            }
            break;
        case(1): // State 1: Grab Frame
            if (!cap_flag)
            {
                grab_frame();
                image_capture_state = 2;
                cap_flag = 1;
                frame_count++;
            }
            break;
        case(2): // State 2: Detect colors
            clr1_temp = detect_color(RED);
            clr2_temp = detect_color(WHITE);
            if (clr1_temp && clr2_temp){
                image_capture_state = 5;
            }
            else if (clr1_temp && !clr2_temp){
                image_capture_state = 3;
            }
            else if (!clr1_temp && clr2_temp){
                image_capture_state = 4;
            }
            else
            {
                temp_sun_found = 0;
                temp_tower_found = 0;
                img_process = 1;
                image_capture_state = 1;
            }
            break;
        case(3): // State 3: Tower detected
            img_process = 1;

```

```

        image_capture_state = 1;          // move back to state 1
        tower_found = 1;
        temp_tower_found = 1;
        temp_sun_found = 0;
        break;
    case(4):          // State 4: Sun detected
        img_process = 1;
        image_capture_state = 1;
        sun_found = 1;
        temp_sun_found = 1;
        temp_tower_found = 0;
        break;
    case(5):          // State 5: Sun and Tower detected
        img_process = 1;
        image_capture_state = 1;
        tower_found = 1;
        temp_tower_found = 1;
        sun_found = 1;
        temp_sun_found = 1;
        break;
    }
}

// This function moves mirror 1 to the appropriate position. It takes an input
mov_camera_state
// which tells the function which state it is currently in.
void task_mov_camera(char mov_cam_state)
{
    unsigned int temp3;
    switch (mov_cam_state)
    {
        case(0):          // State 0: Initialization
            Lowx1 = IMAGE_WIDTH/2 - WIDE_RANGE;
            Highx2 = IMAGE_WIDTH/2 + WIDE_RANGE;
            Lowy1 = IMAGE_HEIGHT/2 - WIDE_RANGE;
            Highy2 = IMAGE_HEIGHT/2 + WIDE_RANGE;
            step = WIDE_STEP;
            mov_camera_state = 1;
            break;
        case(1):          // State 1: Check what has been found in image
            if (img_process)
            {
                img_process = 0;
                if (temp_tower_found && !temp_sun_found)
                {
                    mov_camera_state = 2;
                }
                else if (!temp_tower_found && temp_sun_found)
                {
                    mov_camera_state = 4;
                }
                else if (!temp_tower_found && !temp_sun_found)
                {
                    mov_camera_state = 6;;
                }
                else if (temp_tower_found && temp_sun_found)
                {
                    mov_camera_state = 14;
                }
            }
            break;
        case(2):          // State 2: Only tower found
            if (track_tower)
            {
                mov_camera_state = 3;
            }
            else if (find_tower)
            {
                mov_camera_state = 9;
            }
            else if (move_to_tower)

```

```

{
    mov_camera_state = 7;
}
else if (move_to_sun)
{
    mov_camera_state = 8;
}
else if (find_sun)
{
    mov_camera_state = 10;
}
else if (track_sun)
{
    mov_camera_state = 5;
}
break;
case(3): // State 3: Locate Tower
track_color(RED,0,0);
if (!temp_tower_found)
{
    printf("tower not found \r\n");
}
if (!wide_center_found)
{
    tower_watchdog = 0;
    if (!centeredx && !centeredy){
        move_servo(AZ1,desired_az_camera);
        move_servo(ALT1,desired_alt_camera);
        frame_count = 0;
        mov_camera_state = 1;
        cap_flag = 0;
    }
    else if (centeredx && !centeredy){
        move_servo(ALT1,desired_alt_camera);
        frame_count = 0;
        mov_camera_state = 1;
        cap_flag = 0;
    }
    else if (!centeredx && centeredy){
        move_servo(AZ1,desired_az_camera);
        frame_count = 0;
        mov_camera_state = 1;
        cap_flag = 0;
    }
    else {
        centeredx = 0;
        centeredy = 0;
        wide_center_found = 1;
        step = SLIM_STEP;
        Lowx1 = IMAGE_WIDTH/2 - SLIM_RANGE;
        Highx2 = IMAGE_WIDTH/2 + SLIM_RANGE;
        Lowy1 = IMAGE_HEIGHT/2 - SLIM_RANGE;
        Highy2 = IMAGE_HEIGHT/2 + SLIM_RANGE;
        mov_camera_state = 1;
        cap_flag = 0;
    }
}
else {
    tower_watchdog = 0;
    if (frame_count >= MAX_FRAME_COUNT)
    {
        frame_count_flag = 1;
        if (!centeredx && !centeredy){
            move_servo(AZ1,desired_az_camera);
            move_servo(ALT1,desired_alt_camera);
            frame_count = 0;
            mov_camera_state = 1;
            cap_flag = 0;
        }
        else if (centeredx && !centeredy){
            move_servo(ALT1,desired_alt_camera);

```

```

        frame_count = 0;
        mov_camera_state = 1;
        cap_flag = 0;
    }
    else if (!centeredx && centeredy){
        move_servo(AZ1,desired_az_camera);
        frame_count = 0;
        mov_camera_state = 1;
        cap_flag = 0;
    }
    else {
        printf("Tower centered slim \r\n");
        tower_located = 1;
        frame_count = 0;
        track_tower = 0;
        mov_camera_state = 12;
        Lowx1 = IMAGE_WIDTH/2 - WIDE_RANGE;
        Highx2 = IMAGE_WIDTH/2 + WIDE_RANGE;
        Lowy1 = IMAGE_HEIGHT/2 - WIDE_RANGE;
        Highy2 = IMAGE_HEIGHT/2 + WIDE_RANGE;
        step = WIDE_STEP;
    }
}
else
{
    mov_camera_state = 1;
    cap_flag = 0;
}
}
break;
case(4): // State 4: Only the sun found
    if (track_sun)
    {
        mov_camera_state = 5;
    }
    else if (find_sun)
    {
        mov_camera_state = 10;
    }
    else if (move_to_sun)
    {
        mov_camera_state = 8;
    }
    else if (move_to_tower)
    {
        mov_camera_state = 7;
    }
    else if (find_tower)
    {
        mov_camera_state = 9;
    }
    else if (track_tower)
    {
        mov_camera_state = 3;
    }
    else
    {
    }
    break;
case(5): // State 5: Locate sun
    track_color(WHITE,0,0);
    if (!temp_sun_found)
    {
        printf("sun not found \r\n");
    }
    if (!sun_wide_center_found)
    {
        sun_watchdog = 0;
        if (!centeredx && !centeredy){
            move_servo(AZ1,desired_az_camera);
            move_servo(ALT1,desired_alt_camera);

```

```

        frame_count = 0;
        mov_camera_state = 1;
        cap_flag = 0;
        sun_wide_center_found = 0;
    }
    else if (centeredx && !centeredy){
        move_servo(ALT1,desired_alt_camera);
        frame_count = 0;
        mov_camera_state = 1;
        cap_flag = 0;
        sun_wide_center_found = 0;
    }
    else if (!centeredx && centeredy){
        move_servo(AZ1,desired_az_camera);
        frame_count = 0;
        mov_camera_state = 1;
        cap_flag = 0;
        sun_wide_center_found = 0;
    }
    else {
        centeredx = 0;
        centeredy = 0;
        step = SLIM_STEP;
        Lowx1 = IMAGE_WIDTH/2 - SLIM_RANGE;
        Highx2 = IMAGE_WIDTH/2 + SLIM_RANGE;
        Lowy1 = IMAGE_HEIGHT/2 - SLIM_RANGE;
        Highy2 = IMAGE_HEIGHT/2 + SLIM_RANGE;
        sun_wide_center_found = 1;
        find_sun = 0;
        mov_camera_state = 1;
        cap_flag = 0;
    }
}
else {
    sun_watchdog = 0;
    if (frame_count >= MAX_FRAME_COUNT)
    {
        frame_count_flag = 1;
        if (!centeredx && !centeredy){
            move_servo(AZ1,desired_az_camera);
            move_servo(ALT1,desired_alt_camera);
            frame_count = 0;
            mov_camera_state = 1;
            cap_flag = 0;
        }
        else if (centeredx && !centeredy){
            move_servo(ALT1,desired_alt_camera);
            frame_count = 0;
            mov_camera_state = 1;
            cap_flag = 0;
        }
        else if (!centeredx && centeredy){
            move_servo(AZ1,desired_az_camera);
            frame_count = 0;
            mov_camera_state = 1;
            cap_flag = 0;
        }
        else {
            printf("Sun centered slim \r\n");
            sun_located = 1;
            frame_count = 0;
            track_sun = 0;
            mov_camera_state = 11;
            Lowx1 = IMAGE_WIDTH/2 - WIDE_RANGE;
            Highx2 = IMAGE_WIDTH/2 + WIDE_RANGE;
            Lowy1 = IMAGE_HEIGHT/2 - WIDE_RANGE;
            Highy2 = IMAGE_HEIGHT/2 + WIDE_RANGE;
            step = WIDE_STEP;
        }
    }
}
else

```

```

        {
            mov_camera_state = 1;
            cap_flag = 0;
        }
    }
    break;
case (6): // State 6: Neither tower nor sun found
    if (move_to_tower)
    {
        mov_camera_state = 7;
    }
    else if (find_tower)
    {
        mov_camera_state = 9;
    }
    else if (track_tower)
    {
        mov_camera_state = 3;
    }
    else if (move_to_sun)
    {
        mov_camera_state = 8;
    }
    else if (find_sun)
    {
        mov_camera_state = 10;
    }
    else if (track_sun)
    {
        mov_camera_state = 5;
    }
    break;
case (7): // State 7: Move to known location of tower
    move_servo(AZ1,tower_az);
    move_servo(ALT1,tower_alt);
    if (az1flag == 1 && alt1flag == 1)
    {
        az1flag = 0;
        alt1flag = 0;
        desired_az_camera = tower_az;
        desired_alt_camera = tower_alt;
        mov_camera_state = 16;
    }
    else
    {
        cap_flag = 0;
    }
    break;
case (8): // State 8: Move to known location of sun
    move_servo(AZ1,sun_az);
    move_servo(ALT1,sun_alt);
    if (az1flag == 1 && alt1flag == 1)
    {
        az1flag = 0;
        alt1flag = 0;
        desired_az_camera = sun_az;
        desired_alt_camera = sun_alt;
        mov_camera_state = 15;
    }
    else
    {
        //mov_camera_state = 1;
        cap_flag = 0;
    }
    break;
case (9): // State 9: Scan blind for tower
    if (!tower_found)
    {
        temp3 = alt_camera + WIDE_STEP;
        if (temp3 >= MAX_ALTITUDE)
        {

```

```

        temp3 = MAX_ALTITUDE;
        printf("Max scanning altitude reached \r\n");
    }
    move_servo(ALT1,temp3);
}
else
{
    find_tower = 0;
    track_tower = 1;
}
mov_camera_state = 1;
cap_flag = 0;
break;
case(10): // State 10: Scan blind for the sun
    if (!sun_found)
    {
        temp3 = alt_camera + WIDE_STEP;
        if (temp3 >= MAX_ALTITUDE)
        {
            temp3 = MAX_ALTITUDE;
            printf("Max scanning altitude reached \r\n");
        }
        move_servo(ALT1,temp3);
    }
    else
    {
        find_sun = 0;
        track_sun = 1;
    }
    mov_camera_state = 1;
    cap_flag = 0;
    break;
case(11): // State 11: Sun located
    sun_az = az_1_a2d(az_1_d2a(desired_az_camera) + centeredx_offset);
    sun_alt = alt_1_a2d(alt_1_d2a(desired_alt_camera) + centeredy_offset);

    if (calc_pos)
    {
        mov_camera_state = 13;
    }
    else
    {
        mov_camera_state = 1;
        cap_flag = 0;
    }
    break;
case(12): // State 12: Tower located
    tower_az = az_1_a2d(az_1_d2a(desired_az_camera) + centeredx_offset);
    tower_alt = alt_1_a2d(alt_1_d2a(desired_alt_camera) + centeredy_offset);

    if (calc_pos)
    {
        mov_camera_state = 13;
    }
    else
    {
        mov_camera_state = 1;
        cap_flag = 0;
    }
    break;
case(13): // State 13: Calculation state
    tower_1_x = L*(f_tan(az_calib((az_1_d2a(tower_az)-AZ_ZERO_1))*D2R));
    tower_1_y = L;
    tower_1_z =
(float)isqrt(f_power(tower_1_y,2)+f_power(tower_1_x,2))*f_tan(alt_calib(alt_1_d2a(tower_a
lt) + ALT_ZERO_1)*D2R);

    if(!real_sun)
    {
        sun_1_x = L_sun*(f_tan(az_calib((az_1_d2a(sun_az)-
AZ_ZERO_1))*D2R));

```



```

        if ((az_calib(az_1_d2a(sun_az)-AZ_ZERO_1) > 90) ||
            (az_calib(az_1_d2a(sun_az)-AZ_ZERO_1) < -90))
        {
            sun_1_x *= -1;
        }
        sun_1_y = L_sun;
        sun_1_z =
(float)isqrt(f_power(sun_1_y,2)+f_power(sun_1_x,2))*f_tan(alt_calib((alt_1_d2a(sun_alt) +
ALT_ZERO_1))*D2R);
    }

    printf("===== Camera =====\r\n");
    f_temp1 = az_calib((az_1_d2a(tower_az)-AZ_ZERO_1))*1000;
    f_temp2 = alt_calib((alt_1_d2a(tower_alt)+ALT_ZERO_1))*1000;
    f_temp3 = az_calib((az_1_d2a(sun_az)-AZ_ZERO_1))*1000;
    f_temp4 = alt_calib((alt_1_d2a(sun_alt)+ALT_ZERO_1))*1000;
    printf("tower_az: %f          tower_alt: %f\r\n",f_temp1,f_temp2);
    printf("sun_az: %f          sun_alt: %f\r\n",f_temp3,f_temp4);

    f_temp_az_bisect = (f_temp1 + f_temp3)/2;
    f_temp_alt_bisect = (f_temp2 + f_temp4)/2;

    f_temp1 = tower_1_x*1000;
    f_temp2 = tower_1_y*1000;
    f_temp3 = tower_1_z*1000;
    f_temp4 = sun_1_x*1000;
    f_temp5 = sun_1_y*1000;
    f_temp6 = sun_1_z*1000;

    printf("tower_1_x: %f          tower_1_y: %f          tower_1_z: %f
\r\n",f_temp1,f_temp2,f_temp3);
    printf("sun_1_x: %f          sun_1_y: %f          sun_1_z: %f
\r\n",f_temp4,f_temp5,f_temp6);
    printf("az_bisector: %f          alt_bisector: %f
\r\n",f_temp_az_bisect,f_temp_alt_bisect);
    printf("\r\n");

    mov_camera_state = 1;
    cap_flag = 0;
    mov_mirror_2 = 1;
    mov_mirror_3 = 1;
    mov_mirror_4 = 1;
    calc_pos = 0;
    break;
case(14):    // State 14: Both sun and tower found in the same image
    if (move_to_tower)
    {
        mov_camera_state = 7;
    }
    else if (track_tower)
    {
        mov_camera_state = 3;
    }
    else if (find_tower)
    {
        mov_camera_state = 9;
    }
    else if (move_to_sun)
    {
        mov_camera_state = 8;
    }
    else if (track_sun)
    {
        mov_camera_state = 5;
    }
    else if (find_sun)
    {
        mov_camera_state = 10;
    }
    break;
case(15):    // State 15: Search for sun movement

```

```

        move_to_sun = 0;
        track_sun = 1;
        mov_camera_state = 1;
        cap_flag = 0;
        break;
    case(16): // State 16: Search for tower movement
        move_to_tower = 0;
        track_tower = 1;
        mov_camera_state = 1;
        cap_flag = 0;
        break;
    } // switch (state)
} // task_mov_camera

// This function moves mirror 2 to the appropriate position. It takes an input
mov_mirror_2_state which tells the function which state it is currently in.
void task_mov_mirror_2(char mov_mir_2_state)
{
    float ci;
    float cj;
    float ck;
    float mag_a;
    float mag_b;

    switch (mov_mir_2_state)
    {
    case(0): // initialization state
        if (mov_mirror_2)
        {
            mov_mirror_2_state = 1;
        }
        break;
    case(1): // calculation state -- calculate the position the mirror needs to
be in
        tower_2_x = mirror_2_x + tower_1_x;
        tower_2_y = mirror_2_y + tower_1_y;
        tower_2_z = mirror_2_z + tower_1_z;

        if (f_fabs(tower_2_x) < f_fabs(tower_2_y))
        {
            tower_2_az = (f_atan2(tower_2_x,tower_2_y)*R2D);
        }
        else
        {
            tower_2_az = (90-f_atan2(tower_2_y,tower_2_x)*R2D);
        }

        if (f_fabs(tower_2_z) <
f_fabs(isqrt(f_power(tower_2_x,2)+f_power(tower_2_y,2))))
        {
            tower_2_alt =
(f_atan2(tower_2_z,isqrt(f_power(tower_2_x,2)+f_power(tower_2_y,2)))*R2D);
        }
        else
        {
            tower_2_alt = (90 -
f_atan2(isqrt(f_power(tower_2_x,2)+f_power(tower_2_y,2)),tower_2_z)*R2D);
        }

        if (real_sun)
        {
            sun_2_az = az_calib(az_1_d2a(sun_az) - AZ_ZERO_1);
            sun_2_alt = alt_calib(alt_1_d2a(sun_alt) + ALT_ZERO_1);

            sun_2_x = f_sin(sun_2_az*D2R);
            sun_2_y = f_cos(sun_2_az*D2R);
            sun_2_z = f_tan(sun_2_alt*D2R);
        }
        else
        {

```

```

        sun_2_x = mirror_2_x + sun_1_x;
        sun_2_y = mirror_2_y + sun_1_y;
        sun_2_z = mirror_2_z + sun_1_z;
    }

    mag_a = (float)isqrt(f_power(sun_2_x,2) + f_power(sun_2_y,2) +
f_power(sun_2_z,2));
    mag_b = (float)isqrt(f_power(tower_2_x,2) + f_power(tower_2_y,2) +
f_power(tower_2_z,2));

    ci = mag_a*tower_2_x + mag_b*sun_2_x;
    cj = mag_a*tower_2_y + mag_b*sun_2_y;
    ck = mag_a*tower_2_z + mag_b*sun_2_z;

    if (f_fabs(ci) < f_fabs(cj))
    {
        f_temp = az_2_a2d(f_atan2(ci,cj)*R2D + AZ_ZERO_2);
    }
    else
    {
        f_temp = az_2_a2d(90 - f_atan2(cj,ci)*R2D + AZ_ZERO_2);
    }

    f_temp_az_bisect = (az_2_d2a(f_temp) - AZ_ZERO_2)*1000;

    if (f_temp < 0){
        desired_az_mirror_2 = 0;}
    else if (f_temp > 4000){
        desired_az_mirror_2 = 4000;}
    else{
        desired_az_mirror_2 = f_temp;}

    if (f_fabs(ck) < f_fabs(isqrt(f_power(ci,2) + f_power(cj,2))))
    {
        f_temp = alt_2_a2d(f_atan2(ck,isqrt(f_power(ci,2) +
f_power(cj,2)))*R2D);
    }
    else
    {
        f_temp = alt_2_a2d(90 - f_atan2(isqrt(f_power(ci,2) +
f_power(cj,2)),ck)*R2D);
    }

    if (f_temp < 0){
        desired_alt_mirror_2 = 0;}
    else if (f_temp > 4000){
        desired_alt_mirror_2 = 4000;}
    else{
        desired_alt_mirror_2 = f_temp;}

    f_temp_alt_bisect = (alt_2_d2a(f_temp))*1000;

    f_temp1 = tower_2_az*1000;
    f_temp2 = tower_2_alt*1000;
    f_temp3 = sun_2_az*1000;
    f_temp4 = sun_2_alt*1000;

    printf("===== Mirror 2
=====\\r\\n");
    printf("tower_2_az: %f          tower_2_alt: %f \\r\\n",f_temp1,f_temp2);
    printf("sun_2_az: %f          sun_2_alt: %f \\r\\n",f_temp3,f_temp4);

    f_temp1 = tower_2_x*1000;
    f_temp2 = tower_2_y*1000;
    f_temp3 = tower_2_z*1000;
    f_temp4 = sun_2_x*1000;
    f_temp5 = sun_2_y*1000;
    f_temp6 = sun_2_z*1000;

```

```

        printf("tower_2_x: %f          tower_2_y: %f          tower_2_z:
%f\r\n",f_temp1,f_temp2,f_temp3);
        printf("sun_2_x: %f          sun_2_y: %f          sun_2_z:
%f\r\n",f_temp4,f_temp5,f_temp6);

        printf("az_bisector: %f          alt_bisector: %f
\r\n",f_temp_az_bisect,f_temp_alt_bisect);
        printf("\r\n");

        mov_mirror_2_state = 2;
        break;
    case(2): // move servos state
        move_servo(AZ2,desired_az_mirror_2);
        move_servo(ALT2,desired_alt_mirror_2);
        if (az2flag && alt2flag)
        {
            mov_mirror_2_state = 0;
            mov_mirror_2 = 0;
            az2flag = 0;
            alt2flag = 0;
        }
        break;
    }
}

// This function moves mirror 3 to the appropriate position. It takes an input
mov_mirror_3_state which tells the function which state it is currently in.
void task_mov_mirror_3(char mov_mir_3_state)
{
    float ci;
    float cj;
    float ck;
    float mag_a;
    float mag_b;

    switch (mov_mir_3_state)
    {
    case(0): // initialization state
        if (mov_mirror_3)
        {
            mov_mirror_3_state = 1;
        }
        break;
    case(1): // calculation state -- calculate the position the mirror needs to
be in
        tower_3_x = mirror_3_x + tower_1_x;
        tower_3_y = mirror_3_y + tower_1_y;
        tower_3_z = mirror_3_z + tower_1_z;

        if (f_fabs(tower_3_x) < f_fabs(tower_3_y))
        {
            tower_3_az = f_atan2(tower_3_x,tower_3_y)*R2D;
        }
        else
        {
            tower_3_az = 90-f_atan2(tower_3_y,tower_3_x)*R2D;
        }

        if (f_fabs(tower_3_z) <
f_fabs(isqrt(f_power(tower_3_x,2)+f_power(tower_3_y,2))))
        {
            tower_3_alt =
f_atan2(tower_3_z,isqrt(f_power(tower_3_x,2)+f_power(tower_3_y,2)))*R2D;
        }
        else
        {
            tower_3_alt = 90 -
f_atan2(isqrt(f_power(tower_3_x,2)+f_power(tower_3_y,2)),tower_3_z)*R2D;
        }
    }
}

```

```

if (real_sun)
{
    sun_3_az = az_calib(az_1_d2a(sun_az) - AZ_ZERO_1);
    sun_3_alt = alt_calib(alt_1_d2a(sun_alt) + ALT_ZERO_1);

    sun_3_x = 100.0*f_sin(sun_3_az*D2R);
    sun_3_y = 100.0*f_cos(sun_3_az*D2R);
    sun_3_z = 100.0*f_tan(sun_3_alt*D2R);
}
else
{
    sun_3_x = mirror_3_x + sun_1_x;
    sun_3_y = mirror_3_y + sun_1_y;
    sun_3_z = mirror_3_z + sun_1_z;
}

mag_a = (float)isqrt(f_power(sun_3_x,2) + f_power(sun_3_y,2) +
f_power(sun_3_z,2));
mag_b = (float)isqrt(f_power(tower_3_x,2) + f_power(tower_3_y,2) +
f_power(tower_3_z,2));

ci = mag_a*tower_3_x + mag_b*sun_3_x;
cj = mag_a*tower_3_y + mag_b*sun_3_y;
ck = mag_a*tower_3_z + mag_b*sun_3_z;

if (f_fabs(ci) < f_fabs(cj))
{
    f_temp1 = az_3_a2d((f_atan2(ci,cj)*R2D + AZ_ZERO_3));
}
else
{
    f_temp1 = az_3_a2d((90 - f_atan2(cj,ci)*R2D + AZ_ZERO_3));
}

if (f_fabs(ck) < f_fabs(isqrt(f_power(ci,2) + f_power(cj,2))))
{
    f_temp2 = alt_3_a2d((f_atan2(ck,isqrt(f_power(ci,2) +
f_power(cj,2))*R2D));
}
else
{
    f_temp2 = alt_3_a2d((90 - f_atan2(isqrt(f_power(ci,2) +
f_power(cj,2)),ck)*R2D));
}

f_temp_az_bisect = (az_3_d2a(f_temp1)-AZ_ZERO_3)*1000;
f_temp_alt_bisect = alt_3_d2a(f_temp2)*1000;

if (f_temp1 < 0){
    desired_az_mirror_3 = 0;}
else if (f_temp1 > 4000){
    desired_az_mirror_3 = 4000;}
else{
    desired_az_mirror_3 = f_temp1;}

if (f_temp2 < 0){
    desired_alt_mirror_3 = 0;}
else if (f_temp2 > 4000){
    desired_alt_mirror_3 = 4000;}
else{
    desired_alt_mirror_3 = f_temp2;}

f_temp1 = tower_3_az*1000;
f_temp2 = tower_3_alt*1000;
f_temp3 = sun_3_az*1000;
f_temp4 = sun_3_alt*1000;

printf("===== Mirror 3
=====\\r\\n");

```

```

        printf("tower_3_az: %f          tower_3_alt: %f \r\n",f_temp1,f_temp2);
        printf("sun_3_az: %f          sun_3_alt: %f \r\n",f_temp3,f_temp4);

        f_temp1 = tower_3_x*1000;
        f_temp2 = tower_3_y*1000;
        f_temp3 = tower_3_z*1000;
        f_temp4 = sun_3_x*1000;
        f_temp5 = sun_3_y*1000;
        f_temp6 = sun_3_z*1000;

        printf("tower_3_x: %f          tower_3_y: %f          tower_3_z:
%f\r\n",f_temp1,f_temp2,f_temp3);
        printf("sun_3_x: %f          sun_3_y: %f          sun_3_z:
%f\r\n",f_temp4,f_temp5,f_temp6);
        printf("az_bisector: %f          alt_bisector: %f
\r\n",f_temp_az_bisect,f_temp_alt_bisect);
        printf("\r\n");

        mov_mirror_3_state = 2;
        break;
    case(2): // move servos state
        move_servo(AZ3,desired_az_mirror_3);
        move_servo(ALT3,desired_alt_mirror_3);
        if (az3flag && alt3flag)
        {
            mov_mirror_3_state = 0;
            mov_mirror_3 = 0;
            az3flag = 0;
            alt3flag = 0;
        }
        break;
    }
}

// This function moves mirror 4 to the appropriate position. It takes an input
mov_mirror_4_state which tells the function which state it is currently in.
void task_mov_mirror_4(char mov_mir_4_state)
{
    float ci;
    float cj;
    float ck;
    float mag_a;
    float mag_b;

    switch (mov_mir_4_state)
    {
    case(0): // initialization state
        if (mov_mirror_4)
        {
            mov_mirror_4_state = 1;
        }
        break;
    case(1): // calculation state -- calculate the position the mirror needs to
be in
        tower_4_x = mirror_4_x + tower_1_x;
        tower_4_y = mirror_4_y + tower_1_y;
        tower_4_z = mirror_4_z + tower_1_z;

        if (f_fabs(tower_4_x) < f_fabs(tower_4_y))
        {
            tower_4_az = f_atan2(tower_4_x,tower_4_y)*R2D;
        }
        else
        {
            tower_4_az = 90-f_atan2(tower_4_y,tower_4_x)*R2D;
        }

        if (f_fabs(tower_4_z) <
f_fabs(isqrt(f_power(tower_4_x,2)+f_power(tower_4_y,2))))
        {

```

```

        tower_4_alt =
f_atan2(tower_4_z, isqrt(f_power(tower_4_x,2)+f_power(tower_4_y,2)))*R2D;
    }
    else
    {
        tower_4_alt = 90 -
f_atan2(isqrt(f_power(tower_4_x,2)+f_power(tower_4_y,2)), tower_4_z)*R2D;
    }

    if (real_sun)
    {
        sun_4_az = az_calib(az_1_d2a(sun_az) - AZ_ZERO_1);
        sun_4_alt = alt_calib(alt_1_d2a(sun_alt) + ALT_ZERO_1);

        sun_4_x = 100.0*f_sin(sun_4_az*D2R);
        sun_4_y = 100.0*f_cos(sun_4_az*D2R);
        sun_4_z = 100.0*f_tan(sun_4_alt*D2R);
    }
    else
    {
        sun_4_x = mirror_4_x + sun_1_x;
        sun_4_y = mirror_4_y + sun_1_y;
        sun_4_z = mirror_4_z + sun_1_z;
    }

    mag_a = (float) isqrt(f_power(sun_4_x,2) + f_power(sun_4_y,2) +
f_power(sun_4_z,2));
    mag_b = (float) isqrt(f_power(tower_4_x,2) + f_power(tower_4_y,2) +
f_power(tower_4_z,2));

    ci = mag_a*tower_4_x + mag_b*sun_4_x;
    cj = mag_a*tower_4_y + mag_b*sun_4_y;
    ck = mag_a*tower_4_z + mag_b*sun_4_z;

    if (f_fabs(ci) < f_fabs(cj))
    {
        f_temp = az_4_a2d(f_atan2(ci,cj)*R2D + AZ_ZERO_4);
    }
    else
    {
        f_temp = az_4_a2d(90 - f_atan2(cj,ci)*R2D + AZ_ZERO_4);
    }

    f_temp_az_bisect = (az_4_d2a(f_temp)-AZ_ZERO_4)*1000;

    if (f_temp < 0){
        desired_az_mirror_4 = 0;}
    else if (f_temp > 4000){
        desired_az_mirror_4 = 4000;}
    else{
        desired_az_mirror_4 = f_temp;}

    if (f_fabs(ck) < f_fabs(isqrt(f_power(ci,2) + f_power(cj,2))))
    {
        f_temp = alt_4_a2d(f_atan2(ck, isqrt(f_power(ci,2) +
f_power(cj,2)))*R2D);
    }
    else
    {
        f_temp = alt_4_a2d(90 - f_atan2(isqrt(f_power(ci,2) +
f_power(cj,2)), ck)*R2D);
    }

    f_temp_alt_bisect = alt_4_d2a(f_temp)*1000;

    if (f_temp < 0){
        desired_alt_mirror_4 = 0;}
    else if (f_temp > 4000){
        desired_alt_mirror_4 = 4000;}
    else{
        desired_alt_mirror_4 = f_temp;}

```

```

        f_temp1 = tower_4_az*1000;
        f_temp2 = tower_4_alt*1000;
        f_temp3 = sun_4_az*1000;
        f_temp4 = sun_4_alt*1000;

        printf("===== Mirror 4
=====\\r\\n");
        printf("tower_4_az: %f          tower_4_alt: %f \\r\\n",f_temp1,f_temp2);
        printf("sun_4_az: %f          sun_4_alt: %f \\r\\n",f_temp3,f_temp4);

        f_temp1 = tower_4_x*1000;
        f_temp2 = tower_4_y*1000;
        f_temp3 = tower_4_z*1000;
        f_temp4 = sun_4_x*1000;
        f_temp5 = sun_4_y*1000;
        f_temp6 = sun_4_z*1000;

        printf("tower_4_x: %f          tower_4_y: %f          tower_4_z:
%f\\r\\n",f_temp1,f_temp2,f_temp3);
        printf("sun_4_x: %f          sun_4_y: %f          sun_4_z:
%f\\r\\n",f_temp4,f_temp5,f_temp6);

        printf("az_bisector: %f          alt_bisector: %f
\\r\\n",f_temp_az_bisect,f_temp_alt_bisect);

        printf("\\r\\n");

        mov_mirror_4_state = 2;
        break;
    case (2): // move servos state
        move_servo(AZ4,desired_az_mirror_4);
        move_servo(ALT4,desired_alt_mirror_4);
        if (az4flag && alt4flag)
        {
            mov_mirror_4_state = 0;
            mov_mirror_4 = 0;
            az4flag = 0;
            alt4flag = 0;
        }
        break;
    }
}

// This function moves the servo. Pass the number 1-8 to correspond to which servo, then
the desired position
void move_servo(unsigned char servo_num, unsigned int position)
{
    unsigned char servo_act = servo_num;
    unsigned char data1;
    unsigned char data2;

    position += 969; // 0.492 ms

    switch (servo_num)
    {
        case (0): // servo number 1 selected
            if (az_camera > position+high_range)
            {
                az_camera -= WIDE_STEP;
                if(az_camera == position){
                    az1flag = 1;}
                else{
                    az1flag = 0;}
            }
            else if (az_camera < position-low_range)
            {
                az_camera += WIDE_STEP;
                if(az_camera == position){
                    az1flag = 1;}
                else {

```



```

        az1flag = 0;}
    }
    else if (az_camera > position)
    {
        az_camera--;
        if(az_camera == position){
            az1flag = 1;}
        else{
            az1flag = 0;}
    }
    else if (az_camera < position)
    {
        az_camera++;
        if(az_camera == position){
            az1flag = 1;}
        else {
            az1flag = 0;}
    }
    else {
        az1flag = 1;}

    data1 = az_camera >> 7;
    data2 = az_camera;
    if (data2 > 128)
    {
        data2 -= 128;
    }

    uart1SendChar(start_byte);
    uart1SendChar(device_ID);
    uart1SendChar(command);
    uart1SendChar(servo_act);
    uart1SendChar(data1);
    uart1SendChar(data2);
    break;
case(1):    // servo number 2 selected
    if (alt_camera > position+high_range)
    {
        alt_camera -= WIDE_STEP;
        if(alt_camera == position){
            alt1flag = 1;}
        else {
            alt1flag = 0;}
    }
    else if (alt_camera < position-low_range)
    {
        alt_camera += WIDE_STEP;
        if(alt_camera == position){
            alt1flag = 1;}
        else {
            alt1flag = 0;}
    }
    else if (alt_camera > position)
    {
        alt_camera--;
        if(alt_camera == position){
            alt1flag = 1;}
        else {
            alt1flag = 0;}
    }
    else if (alt_camera < position)
    {
        alt_camera++;
        if(alt_camera == position){
            alt1flag = 1;}
        else {
            alt1flag = 0;}
    }
    else {
        alt1flag = 1;}

```

```

data1 = alt_camera >> 7;
data2 = alt_camera;
if (data2 > 128)
{
    data2 -= 128;
}

uart1SendChar(start_byte);
uart1SendChar(device_ID);
uart1SendChar(command);
uart1SendChar(servo_act);
uart1SendChar(data1);
uart1SendChar(data2);
break;
case (2): // servo number 3 selected
if (az_mirror_2 > position+high_range)
{
    az_mirror_2 -= WIDE_STEP;
    if(az_mirror_2 == position){
        az2flag = 1;}
    else {
        az2flag = 0;}
}
else if (az_mirror_2 < position-low_range)
{
    az_mirror_2 += WIDE_STEP;
    if(az_mirror_2 == position){
        az2flag = 1;}
    else {
        az2flag = 0;}
}
else if (az_mirror_2 > position)
{
    az_mirror_2--;
    if(az_mirror_2 == position){
        az2flag = 1;}
    else {
        az2flag = 0;}
}
else if (az_mirror_2 < position)
{
    az_mirror_2++;
    if(az_mirror_2 == position){
        az2flag = 1;}
    else {
        az2flag = 0;}
}
else {
    az2flag = 1;}

data1 = az_mirror_2 >> 7;
data2 = az_mirror_2;
if (data2 > 128)
{
    data2 -= 128;
}

uart1SendChar(start_byte);
uart1SendChar(device_ID);
uart1SendChar(command);
uart1SendChar(servo_act);
uart1SendChar(data1);
uart1SendChar(data2);
delayMS(DELAY_TIME);
break;
case (3): // servo number 4 selected
if (alt_mirror_2 > position+high_range)
{
    alt_mirror_2 -= WIDE_STEP;
    if(alt_mirror_2 == position){
        alt2flag = 1;}

```

```

        else {
            alt2flag = 0;}
    }
    else if (alt_mirror_2 < position-low_range)
    {
        alt_mirror_2 += WIDE_STEP;
        if(alt_mirror_2 == position){
            alt2flag = 1;}
        else {
            alt2flag = 0;}
    }
    else if (alt_mirror_2 > position)
    {
        alt_mirror_2--;
        if(alt_mirror_2 == position){
            alt2flag = 1;}
        else {
            alt2flag = 0;}
    }
    else if (alt_mirror_2 < position)
    {
        alt_mirror_2++;
        if(alt_mirror_2 == position){
            alt2flag = 1;}
        else {
            alt2flag = 0;}
    }
    else {
        alt2flag = 1;}

    data1 = alt_mirror_2 >> 7;
    data2 = alt_mirror_2;
    if (data2 > 128)
    {
        data2 -= 128;
    }

    uart1SendChar(start_byte);
    uart1SendChar(device_ID);
    uart1SendChar(command);
    uart1SendChar(servo_act);
    uart1SendChar(data1);
    uart1SendChar(data2);
    delayMS(DELAY_TIME);
    break;
case (4):    // servo number 5 selected
    if (az_mirror_3 > position+high_range)
    {
        az_mirror_3 -= WIDE_STEP;
        if(az_mirror_3 == position){
            az3flag = 1;}
        else {
            az3flag = 0;}
    }
    else if (az_mirror_3 < position-low_range)
    {
        az_mirror_3 += WIDE_STEP;
        if(az_mirror_3 == position){
            az3flag = 1;}
        else {
            az3flag = 0;}
    }
    else if (az_mirror_3 > position)
    {
        az_mirror_3--;
        if(az_mirror_3 == position){
            az3flag = 1;}
        else {
            az3flag = 0;}
    }
    else if (az_mirror_3 < position)

```

```

{
    az_mirror_3++;
    if(az_mirror_3 == position){
        az3flag = 1;}
    else {
        az3flag = 0;}
}
else {
    az3flag = 1;}

data1 = az_mirror_3 >> 7;
data2 = az_mirror_3;
if (data2 > 128)
{
    data2 -= 128;
}

uart1SendChar(start_byte);
uart1SendChar(device_ID);
uart1SendChar(command);
uart1SendChar(servo_act);
uart1SendChar(data1);
uart1SendChar(data2);
delayMS(DELAY_TIME);
break;
case(5): // servo number 6 selected
if (alt_mirror_3 > position+high_range)
{
    alt_mirror_3 -= WIDE_STEP;
    if(alt_mirror_3 == position){
        alt3flag = 1;}
    else {
        alt3flag = 0;}
}
else if (alt_mirror_3 < position-low_range)
{
    alt_mirror_3 += WIDE_STEP;
    if(alt_mirror_3 == position){
        alt3flag = 1;}
    else {
        alt3flag = 0;}
}
else if (alt_mirror_3 > position)
{
    alt_mirror_3--;
    if(alt_mirror_3 == position){
        alt3flag = 1;}
    else {
        alt3flag = 0;}
}
else if (alt_mirror_3 < position)
{
    alt_mirror_3++;
    if(alt_mirror_3 == position){
        alt3flag = 1;}
    else {
        alt3flag = 0;}
}
else {
    alt3flag = 1;}

data1 = alt_mirror_3 >> 7;
data2 = alt_mirror_3;
if (data2 > 128)
{
    data2 -= 128;
}

uart1SendChar(start_byte);
uart1SendChar(device_ID);
uart1SendChar(command);

```

```

    uart1SendChar(servo_act);
    uart1SendChar(data1);
    uart1SendChar(data2);
    delayMS(DELAY_TIME);
    break;
case(6): // servo number 7 selected
    if (az_mirror_4 > position+high_range)
    {
        az_mirror_4 -= WIDE_STEP;
        if(az_mirror_4 == position){
            az4flag = 1;}
        else {
            az4flag = 0;}
    }
    else if (az_mirror_4 < position-low_range)
    {
        az_mirror_4 += WIDE_STEP;
        if(az_mirror_4 == position){
            az4flag = 1;}
        else {
            az4flag = 0;}
    }
    else if (az_mirror_4 > position)
    {
        az_mirror_4--;
        if(az_mirror_4 == position){
            az4flag = 1;}
        else {
            az4flag = 0;}
    }
    else if (az_mirror_4 < position)
    {
        az_mirror_4++;
        if(az_mirror_4 == position){
            az4flag = 1;}
        else {
            az4flag = 0;}
    }
    else {
        az4flag = 1;}

    data1 = az_mirror_4 >> 7;
    data2 = az_mirror_4;
    if (data2 > 128)
    {
        data2 -= 128;
    }

    uart1SendChar(start_byte);
    uart1SendChar(device_ID);
    uart1SendChar(command);
    uart1SendChar(servo_act);
    uart1SendChar(data1);
    uart1SendChar(data2);
    delayMS(DELAY_TIME);
    break;
case(7): // servo number 8 selected
    if (alt_mirror_4 > position+high_range)
    {
        alt_mirror_4 -= WIDE_STEP;
        if(alt_mirror_4 == position){
            alt4flag = 1;}
        else {
            alt4flag = 0;}
    }
    else if (alt_mirror_4 < position-low_range)
    {
        alt_mirror_4 += WIDE_STEP;
        if(alt_mirror_4 == position){
            alt4flag = 1;}
        else {

```

```

        alt4flag = 0;}
    }
    else if (alt_mirror_4 > position)
    {
        alt_mirror_4--;
        if(alt_mirror_4 == position){
            alt4flag = 1;}
        else {
            alt4flag = 0;}
    }
    else if (alt_mirror_4 < position)
    {
        alt_mirror_4++;
        if(alt_mirror_4 == position){
            alt4flag = 1;}
        else {
            alt4flag = 0;}
    }
    else {
        alt4flag = 1;}

    data1 = alt_mirror_4 >> 7;
    data2 = alt_mirror_4;
    if (data2 > 128)
    {
        data2 -= 128;
    }

    uart1SendChar(start_byte);
    uart1SendChar(device_ID);
    uart1SendChar(command);
    uart1SendChar(servo_act);
    uart1SendChar(data1);
    uart1SendChar(data2);
    delayMS(DELAY_TIME);
    break;
}

}

char detect_color(int COLOR_TO_TRACK)
{
    int num_blobs = 0;
    int ii = 0;
    unsigned int target_centerx = 0;
    unsigned int target_centery = 0;

    set_color_bin(COLOR_TO_TRACK);
    num_blobs = vblob((unsigned char*)FRAME_BUF, (unsigned char*)FRAME_BUF3, 0);
    //identifies a blob (largest blob is 0) from bin 2
    if (num_blobs >= 1 && ((blobx2[0] - blobx1[0]) > MIN_BLOB_WIDTH) && ((bloby2[0] -
bloby1[0]) > MIN_BLOB_HEIGHT))
    {
        target_centerx = (blobx1[0] + blobx2[0])/2;
        target_centery = (bloby1[0] + bloby2[0])/2;
        if ((target_centerx < BORDER) || (target_centerx > IMAGE_WIDTH - BORDER)
|| (target_centery < BORDER) || (target_centery > IMAGE_HEIGHT - BORDER))
        {
            printf("break beacuse of border \r\n");
            return 0;
        }
        switch(COLOR_TO_TRACK)
        {
            case (RED):
                if (!start_flag_red){
                    for (ii = 0; ii < MAX_AVE_NUM; ii++){
                        img_bufx_red[ii] = target_centerx;
                        img_bufy_red[ii] = target_centery;
                    }
                    start_flag_red = 1;
                }
            else {

```

```

        target_centerx_red = target_centerx;
        target_centery_red = target_centery;
        *img_ptrx_red = target_centerx_red;
        *img_ptry_red = target_centery_red;
        img_ptrx_red++;
        img_ptry_red++;
        img_ptr_count_red++;
        if (img_ptr_count_red == MAX_AVE_NUM){
            img_ptr_count_red = 0;
            img_ptrx_red = img_bufx_red;
            img_ptry_red = img_bufy_red;}
    }
    break;
case (WHITE):
    if (!start_flag_white){
        for (ii = 0; ii < MAX_AVE_NUM; ii++){
            img_bufx_white[ii] = target_centerx;
            img_bufy_white[ii] = target_centery;
        }
        start_flag_white = 1;
    }
    else {
        target_centerx_white = target_centerx;
        target_centery_white = target_centery;
        *img_ptrx_white = target_centerx_white;
        *img_ptry_white = target_centery_white;
        img_ptrx_white++;
        img_ptry_white++;
        img_ptr_count_white++;
        if (img_ptr_count_white == MAX_AVE_NUM){
            img_ptr_count_white = 0;
            img_ptrx_white = img_bufx_white;
            img_ptry_white = img_bufy_white;}
    }
    break;
case (BLACK):
    if (!start_flag_black){
        for (ii = 0; ii < MAX_AVE_NUM; ii++){
            img_bufx_black[ii] = target_centerx;
            img_bufy_black[ii] = target_centery;
        }
        start_flag_black = 1;
    }
    else {
        target_centerx_black = target_centerx;
        target_centery_black = target_centery;
        *img_ptrx_black = target_centerx_black;
        *img_ptry_black = target_centery_black;
        img_ptrx_black++;
        img_ptry_black++;
        img_ptr_count_black++;
        if (img_ptr_count_black == MAX_AVE_NUM){
            img_ptr_count_black = 0;
            img_ptrx_black = img_bufx_black;
            img_ptry_black = img_bufy_black;}
    }
    break;
}
return(1);
}
else
{
    return(0);
}
}

void track_color(char COLOR_TO_TRACK, char COLOR_TO_TRACK2, char option)
{
    int TARGET_COLOR_X = 0;
    int TARGET_COLOR_Y = 0;
    int TARGET_COLOR_X2 = 0;

```

```

int TARGET_COLOR_Y2 = 0;
int tempx = 0;
int tempy = 0;
int i = 0;
switch(COLOR_TO_TRACK)
{
    case (RED):
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempx += img_bufx_red[i];}
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempy += img_bufy_red[i];}
        TARGET_COLOR_X = tempx/MAX_AVE_NUM;
        TARGET_COLOR_Y = tempy/MAX_AVE_NUM;
        break;
    case (WHITE):
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempx += img_bufx_white[i];}
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempy += img_bufy_white[i];}
        TARGET_COLOR_X = tempx/MAX_AVE_NUM;
        TARGET_COLOR_Y = tempy/MAX_AVE_NUM;
        break;
    case (BLACK):
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempx += img_bufx_black[i];}
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempy += img_bufy_black[i];}
        TARGET_COLOR_X = tempx/MAX_AVE_NUM;
        TARGET_COLOR_Y = tempy/MAX_AVE_NUM;
        break;
}

tempx = 0;
tempy = 0;

switch(COLOR_TO_TRACK2)
{
    case (RED):
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempx += img_bufx_red[i];}
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempy += img_bufy_red[i];}
        TARGET_COLOR_X2 = tempx/MAX_AVE_NUM;
        TARGET_COLOR_Y2 = tempy/MAX_AVE_NUM;
        break;
    case (WHITE):
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempx += img_bufx_white[i];}
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempy += img_bufy_white[i];}
        TARGET_COLOR_X2 = tempx/MAX_AVE_NUM;
        TARGET_COLOR_Y2 = tempy/MAX_AVE_NUM;
        break;
    case (BLACK):
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempx += img_bufx_black[i];}
        for (i = 0; i < MAX_AVE_NUM; i++){
            tempy += img_bufy_black[i];}
        TARGET_COLOR_X2 = tempx/MAX_AVE_NUM;
        TARGET_COLOR_Y2 = tempy/MAX_AVE_NUM;
        break;
}

tempx = 0;
tempy = 0;

switch(COLOR_TO_TRACK)
{
    case (RED): // one color found
        if (wide_center_found){

```



```

        if ((TARGET_COLOR_X <= (IMAGE_WIDTH/2-WIDE_RANGE)) ||
(TARGET_COLOR_X >= (IMAGE_WIDTH/2+WIDE_RANGE))){
            wide_center_found = 0;
            step = WIDE_STEP;
            Lowx1 = IMAGE_WIDTH/2 - WIDE_RANGE;
            Highx2 = IMAGE_WIDTH/2 + WIDE_RANGE;
            Lowy1 = IMAGE_HEIGHT/2 - WIDE_RANGE;
            Highy2 = IMAGE_HEIGHT/2 + WIDE_RANGE;
        }
        else if ((TARGET_COLOR_Y <= (IMAGE_HEIGHT/2-WIDE_RANGE)) ||
(TARGET_COLOR_Y >= (IMAGE_HEIGHT/2+WIDE_RANGE))){
            wide_center_found = 0;
            step = WIDE_STEP;
            Lowx1 = IMAGE_WIDTH/2 - WIDE_RANGE;
            Highx2 = IMAGE_WIDTH/2 + WIDE_RANGE;
            Lowy1 = IMAGE_HEIGHT/2 - WIDE_RANGE;
            Highy2 = IMAGE_HEIGHT/2 + WIDE_RANGE;
        }

        if (!frame_count_flag){
            break;}
        else {
            frame_count_flag = 0;}
    }

    if(TARGET_COLOR_X <= Lowx1 || TARGET_COLOR_X >= Highx2)
    {
        centeredx = 0;
        if (TARGET_COLOR_X <= Lowx1){
            tempx = desired_az_camera - step;
            if (tempx < 0){
                desired_az_camera = 0;}
            else{
                desired_az_camera -= step;}
        }
        else if (TARGET_COLOR_X >= Highx2){
            tempx = desired_az_camera + step;
            if (tempx > 4000){
                desired_az_camera = 4000;}
            else{
                desired_az_camera += step;}
        }
    }
    else
    {
        centeredx_offset = -IMAGE_WIDTH*0.5 + TARGET_COLOR_X; //
pixel count
        centeredx_offset *= IMAGE_PIX_RATIO; // angle
        centeredx = 1;
    }

    if(TARGET_COLOR_Y <= Lowy1 || TARGET_COLOR_Y >= Highy2)
    {
        // origin is in top left of frame
        centeredy = 0;
        if (TARGET_COLOR_Y <= Lowy1){
            tempy = desired_alt_camera + step;
            if (tempy > 4000){
                desired_alt_camera = 4000;}
            else{
                desired_alt_camera += step;}
        }
        else if (TARGET_COLOR_Y >= Highy2){
            tempy = desired_alt_camera - step;
            if (tempy < 0){
                desired_alt_camera = 0;}
            else{
                desired_alt_camera -= step;}
        }
    }
    else
    {

```

```

                                centeredy_offset = -TARGET_COLOR_Y + IMAGE_HEIGHT*0.5; //
pixel count                                centeredy_offset *= IMAGE_PIX_RATIO;          // angle
                                centeredy = 1;
                                }
                                printf("Red Blob Center X: %d, Blob Center Y: %d \r\n",
TARGET_COLOR_X, TARGET_COLOR_Y);
                                break;
                                case(WHITE): // one color found
                                if (sun_wide_center_found){
                                    if ((TARGET_COLOR_X <= (IMAGE_WIDTH/2-WIDE_RANGE)) ||
(TARGET_COLOR_X >= (IMAGE_WIDTH/2+WIDE_RANGE))){
                                        sun_wide_center_found = 0;
                                        step = WIDE_STEP;
                                        Lowx1 = IMAGE_WIDTH/2 - WIDE_RANGE;
                                        Highx2 = IMAGE_WIDTH/2 + WIDE_RANGE;
                                        Lowy1 = IMAGE_HEIGHT/2 - WIDE_RANGE;
                                        Highy2 = IMAGE_HEIGHT/2 + WIDE_RANGE;
                                    }
                                    else if ((TARGET_COLOR_Y <= (IMAGE_HEIGHT/2-WIDE_RANGE)) ||
(TARGET_COLOR_Y >= (IMAGE_HEIGHT/2+WIDE_RANGE))){
                                        sun_wide_center_found = 0;
                                        step = WIDE_STEP;
                                        Lowx1 = IMAGE_WIDTH/2 - WIDE_RANGE;
                                        Highx2 = IMAGE_WIDTH/2 + WIDE_RANGE;
                                        Lowy1 = IMAGE_HEIGHT/2 - WIDE_RANGE;
                                        Highy2 = IMAGE_HEIGHT/2 + WIDE_RANGE;
                                    }
                                }

                                if (!frame_count_flag){
                                    break;}
                                else {
                                    frame_count_flag = 0;}
                                }

                                if(TARGET_COLOR_X <= Lowx1 || TARGET_COLOR_X >= Highx2)
                                {
                                    centeredx = 0;
                                    if (TARGET_COLOR_X <= Lowx1){
                                        tempx = desired_az_camera - step;
                                        if (tempx < 0){
                                            desired_az_camera = 0;}
                                        else{
                                            desired_az_camera -= step;}
                                    }
                                    else if (TARGET_COLOR_X >= Highx2){
                                        tempx = desired_az_camera + step;
                                        if (tempx > 4000){
                                            desired_az_camera = 4000;}
                                        else{
                                            desired_az_camera += step;}
                                    }
                                }
                                else
                                {
                                    centeredx_offset = -IMAGE_WIDTH*0.5 + TARGET_COLOR_X; //
pixel count                                    centeredx_offset *= IMAGE_PIX_RATIO;          // angle
                                    centeredx = 1;
                                }

                                if(TARGET_COLOR_Y <= Lowy1 || TARGET_COLOR_Y >= Highy2)
                                {
                                    // origin is in top left of frame
                                    centeredy = 0;
                                    if (TARGET_COLOR_Y <= Lowy1){
                                        tempy = desired_alt_camera + step;
                                        if (tempy > 4000){
                                            desired_alt_camera = 4000;}
                                        else{
                                            desired_alt_camera += step;}
                                    }
                                }

```

```

        else if (TARGET_COLOR_Y >= Highy2){
            tempy = desired_alt_camera - step;
            if (tempy < 0){
                desired_alt_camera = 0;}
            else{
                desired_alt_camera -= step;}
        }
    }
    else
    {
        centeredy_offset = -TARGET_COLOR_Y + IMAGE_HEIGHT*0.5; //
pixel count        centeredy_offset *= IMAGE_PIX_RATIO;           // angle
        centeredy = 1;
    }
    printf("White Blob Center X: %d, Blob Center Y: %d \r\n",
TARGET_COLOR_X, TARGET_COLOR_Y);
    break;
}

}

void set_color_bin(unsigned char color_code)
{
    unsigned int ii = 0;
    switch(color_code)
    {
        case(WHITE):
            for(ii = 0; ii<MAX_COLORS; ii++)
            {
                ymax[ii] = 255;
                ymin[ii] = 254;
                umax[ii] = 131;
                umin[ii] = 123;
                vmax[ii] = 131;
                vmin[ii] = 123;
            }
            break;
        case(BLACK):
            for(ii = 0; ii<MAX_COLORS; ii++)
            {
                ymax[ii] = 40;
                ymin[ii] = 0;
                umax[ii] = 132;
                umin[ii] = 123;
                vmax[ii] = 132;
                vmin[ii] = 123;
            }
            break;
        case(RED):
            for(ii = 0; ii<MAX_COLORS; ii++)
            {
                // Color ranges for tower outside
                if (real_sun)
                {
                    ymax[ii] = 195;
                    ymin[ii] = 48;
                    umax[ii] = 130;
                    umin[ii] = 99;
                    vmax[ii] = 217;
                    vmin[ii] = 150;
                }
                // Color ranges for tower for inside
                else
                {
                    ymax[ii] = 150;
                    ymin[ii] = 50;
                    umax[ii] = 125;
                    umin[ii] = 50;
                    vmax[ii] = 255;
                    vmin[ii] = 150;
                }
            }
        }
    }
}

```

```

        }
        break;
    }
}

// Main Program
void heliostat()
{
    while(stop_flag == 0)
    {
        task_mastermind(mm_state);
        task_UI(UI_state);
        task_image_capture(image_capture_state);
        task_mov_camera(mov_camera_state);
        task_mov_mirror_2(mov_mirror_2_state);
        task_mov_mirror_3(mov_mirror_3_state);
        task_mov_mirror_4(mov_mirror_4_state);
    }
    set_color_bin(RED);
    printf("Program Stopped. \r\n");

    return;
}

void task_mastermind(char master_state)
{
    char ch;
    switch(master_state)
    {
        case(0): // State 0: Initialization
            startup();
            mm_state = 1;
            break;
        case(1):
            if (UI_done_flag)
            {
                mm_state = 2;
                move_to_tower = 1;
            }
            break;
        case(2): // State 2: Search for tower
            if (tower_located)
            {
                if (use_gps_flag || use_sun_pos_flag)
                {
                    move_to_sun = 1;
                    AWB_on();
                }
                else
                {
                    find_sun = 1;
                }
                mm_state = 3;
                find_tower = 0;
                tower_located = 0;
            }
            break;
        case(3): // State 3: Searching for sun
            if (sun_wide_center_found)
            {
                if (!first_time)
                {
                    AWB_on();
                    first_time = 1;
                    printf("Sun centered wide \r\n");
                }
                else
                {
                    // do nothing
                }
                //sun_wide_center_found = 0;
            }
    }
}

```

```

    }
    if (sun_located)
    {
        mm_state = 4;
        move_to_tower = 1;
        AWB_off();

        find_sun = 0;
        sun_located = 0;
        calc_pos = 1;
    }
    break;
case (4): // State 4: looking for tower
    if (tower_located)
    {
        mm_state = 5;
        calc_pos = 1;
        move_to_tower = 0;
    }
    break;
case (5): // State 5: User input
    if (!calc_pos && !mov_mirror_2 && !mov_mirror_3 && !mov_mirror_4)
    {
        printf("Press 't' to refind tower, 's' to refind sun, 'e'
to exit: ");

        ch = getch();
        printf("%c \r\n", ch);
        switch(ch)
        {
            case 't':
                move_to_tower = 1;
                mm_state = 4;
                AWB_off();
                done_flag = 0;
                mm_count = 0;
                tower_located = 0;
                wide_center_found = 0;
                break;
            case 's':
                move_to_sun = 1;
                AWB_on();
                sun_wide_center_found = 0;
                move_to_tower = 0;
                find_tower = 0;
                track_tower = 0;
                mm_state = 6;
                first_time = 0;
                break;
            case 'e':
                stop_flag = 1;
                break;
        }
    }
    break;
case (6): // retrack sun
    if (sun_wide_center_found)
    {
        if (!first_time)
        {
            AWB_on();
            printf("Sun centered wide \r\n");
            first_time = 1;
        }
        else
        {
            // do nothing
        }
    }
    if (sun_located)
    {
        mm_state = 5;

```

```

        find_sun = 0;
        sun_located = 0;
        calc_pos = 1;
    }
    break;
} // switch (state)
} // task_mastermind

void startup(void)
{
    int ijk = 0;
    for (ijk = 0; ijk < MAX_AVE_NUM; ijk++)
    {
        img_bufx_red[ijk] = 0;
        img_bufy_red[ijk] = 0;
        img_bufx_white[ijk] = 0;
        img_bufy_white[ijk] = 0;
        img_bufx_black[ijk] = 0;
        img_bufy_black[ijk] = 0;
    }

    az_camera = 969;
    alt_camera = 969;
    az_mirror_2 = 969;
    alt_mirror_2 = 969;
    az_mirror_3 = 969;
    alt_mirror_3 = 969;
    az_mirror_4 = 969;
    alt_mirror_4 = 969;

    // Zeroes, in degrees
    AZ_ZERO_1 = 90;
    ALT_ZERO_1 = 0;
    AZ_ZERO_2 = 87;
    ALT_ZERO_2 = 0;
    AZ_ZERO_3 = 87;
    ALT_ZERO_3 = 0;
    AZ_ZERO_4 = 88;
    ALT_ZERO_4 = 0;
    printf("AZ_ZERO_1: %f, AZ_ZERO_2: %f, AZ_ZERO_3: %f, AZ_ZERO_4: %f\n",
    AZ_ZERO_1, AZ_ZERO_2, AZ_ZERO_3, AZ_ZERO_4);

    // Initialize the servo positions
    desired_az_camera = az_1_a2d(AZ_ZERO_1);
    desired_alt_camera = 9*16;
    desired_az_mirror_2 = az_2_a2d(AZ_ZERO_2);
    desired_alt_mirror_2 = 9*16;
    desired_az_mirror_3 = az_3_a2d(AZ_ZERO_3);
    desired_alt_mirror_3 = 9*16;
    desired_az_mirror_4 = az_4_a2d(AZ_ZERO_4);
    desired_alt_mirror_4 = 9*16;

    // Move the servos until the desired initial positions are reached
    while((az1flag && alt1flag &&
    az2flag && alt2flag &&
    az3flag && alt3flag &&
    az4flag && alt4flag) == 0){
        move_servo(AZ1,desired_az_camera);
        move_servo(ALT1,desired_alt_camera);
        move_servo(AZ2,desired_az_mirror_2);
        move_servo(ALT2,desired_alt_mirror_2);
        move_servo(AZ3,desired_az_mirror_3);
        move_servo(ALT3,desired_alt_mirror_3);
        move_servo(AZ4,desired_az_mirror_4);
        move_servo(ALT4,desired_alt_mirror_4);
    }
}

// Turn Auto-White Balancing, Auto Gain Control, and Auto Exposure Control on
void AWB_on(void)
{

```

```

        i2c_info[0] = 0x13;
        i2c_info[1] = 0xC0 + 0x07;
        i2cwrite(0x21, (unsigned char *)i2c_info, 1, SCCB_ON);
        delayMS(10);
    }

    // Turn Auto-White Balancing off
    void AWB_off(void)
    {
        int ix = 0;
        for (ix = 0; ix < 0xC7; ix++)
        {
            i2c_info[0] = ix;
            i2c_info[1] = data_store[ix];
            i2cwrite(0x21, (unsigned char *)i2c_info, 1, SCCB_ON);
        }
    }

    // Initialize AWB, AGC, and AEC
    void AWB_init(void)
    {
        if (!real_sun)
        {
            i2c_info[1] = 0xC0 + (0 & 0x07);           // turns AWB, AGC, AEC off
        }
        else
        {
            i2c_info[1] = 0xC0 + (0x05 & 0x07);         // turns AWB off, leaves AGC
        }
        and AEC on
        i2c_info[0] = 0x13;
        i2cwrite(0x21, (unsigned char *)i2c_info, 1, SCCB_ON);
    }

    void task_UI(char state_UI)
    {
        unsigned int ix;
        unsigned char ch;

        switch(state_UI)
        {
            case(0): // initialization
                printf("Welcome to the Autonomous Heliostat! \r\n");
                printf(" Version 1.0 -- Joel Masters, 2/3/2011 \r\n");
                delayMS(1000);
                printf("Select an Option: \r\n Press 1 to track the sun \r\n Press
2 to track a light source \r\n");
                UI_state = 1;
                break;
            case(1): // get option char
                ch = getch();
                switch(ch)
                {
                    case('1'): // Sun
                        printf("Option 1 selected. Tracking the Sun. \r\n");
                        printf("Use GPS? (Y/N): ");
                        real_sun = 1;
                        UI_state = 2;
                        option_flag = 1;
                        break;
                    case('2'): // Light Source
                        printf("Option 2 selected. Tracking a light source.
\r\n");
                        printf("Enter the depth from the camera to the
tower: ");

                        real_sun = 0;
                        UI_state = 3;
                        tower_num = 1;
                        option_flag = 2;

```

```

                                break;
        }
        break;
case(2): // get sun option char
    ch = getch();
    printf("%c \r\n",ch);
    switch(ch)
    {
        case('Y'):
        case('y'):
            printf("Use GPS Selected. \r\n");
            printf("Enter longitude: ");
            use_gps_flag = 1;
            gps_num = 1;
            UI_state = 3;
            break;
        case('N'):
        case('n'):
            printf("Use GPS Not Selected. \r\n");
            printf("Enter the depth from the camera to the

tower: ");

            use_gps_flag = 0;
            UI_state = 3;
            tower_num = 1;
            break;
        default:
            printf("Invalid character. \r\n");
            break;
    }
    break;
case(3): // get values
    while (!done_num_flag)
    {
        ch = getch();
        if ((ch >= 48) && (ch <= 57)) // number 0-9
        {
            ch = ch-48;
            if (data_count >= 4)
            {
                // do nothing
            }
            else
            {
                *buf_ptr = ch;
                buf_ptr++;
                printf("%x",ch);
                data_count++;
            }
        }
        else if (ch == 8) // backspace
        {
            if (data_count > 0)
            {
                printf("\b");
                printf(" ");
                printf("\b");
                data_count--;
                buf_ptr--;
                *buf_ptr = 0;
            }
            if (data_count == 0 && neg_flag == 1)
            {
                printf("\b");
                printf(" ");
                printf("\b");
                neg_flag = 0;
            }
        }
        else if (ch == 45) // negative sign
        {

```



```

        if (data_count == 0 && !neg_flag)
        {
            neg_flag = 1;
            printf("-");
        }
    }
    else if (ch == 13)    // enter
    {
        if (data_count > 0)
        {
            for (ix = data_count; ix > 0; ix--)
            {
                switch(ix)
                {
                    case (1):
                        UI_temp_calc +=
* (buf_ptr-ix);
                        break;
                    case (2):
                        UI_temp_calc +=
* (buf_ptr-ix)*10;
                        break;
                    case (3):
                        UI_temp_calc +=
* (buf_ptr-ix)*100;
                        break;
                    case (4):
                        UI_temp_calc +=
* (buf_ptr-ix)*1000;
                        break;
                    default:
                        printf("ix = %d",ix);
                        break;
                }
            }
            if (UI_temp_calc >= 0)
            {
                if (gps_num)
                {
                    switch(gps_num)
                    {
                        case (1):    //
longitude
                                UI_state = 4;
                                break;
                        case (2):    //
latitude
                                UI_state = 5;
                                break;
                        case (3):    //
angle
                                UI_state = 6;
                                break;
                        case (4):    //
month
                                UI_state = 7;
                                break;
                        case (5):    // day
                                UI_state = 8;
                                break;
                        case (6):    // year
                                UI_state = 9;
                                break;
                        case (7):    //
hours
                                UI_state = 10;
                                break;
                        case (8):    //
minutes
                                UI_state = 11;
                                break;
                    }
                }
            }
        }
    }
}

```

```

        }
        data_count = 0;
        buf_ptr -= data_count;
        done_num_flag = 1;
        printf("\r\n");
    }
    if (tower_num) // tower distance
    {
        switch(tower_num)
        {
            case(1): // tx
                UI_state = 12;
                break;
            case(2): // ty
                UI_state = 13;
                break;
            case(3): // tz
                UI_state = 14;
                break;
        }
        data_count = 0;
        buf_ptr -= data_count;
        done_num_flag = 1;
        printf("\r\n");
    }
    if (sun_num) // sun distance
    {
        switch(sun_num)
        {
            case(1): // sx
                UI_state = 15;
                break;
            case(2): // sy
                UI_state = 16;
                break;
            case(3): // sz
                UI_state = 17;
                break;
        }
        data_count = 0;
        buf_ptr -= data_count;
        done_num_flag = 1;
        printf("\r\n");
    }
}

}

}

break;
case(4): // longitude
    if (neg_flag)
    {
        gps_long = -1*UI_temp_calc;
        neg_flag = 0;
    }
    else
    {
        gps_long = UI_temp_calc;
    }

    UI_temp_calc = 0;
    gps_num = 2; // transition to latitude
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter latitude: ");
    break;
case(5): // latitude
    if (neg_flag)
    {
        gps_lat = -1*UI_temp_calc;
        neg_flag = 0;
    }

```

```

    }
    else
    {
        gps_lat = UI_temp_calc;
    }

    UI_temp_calc = 0;
    gps_num = 3;
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter angle in degrees from North, CW positive: ");
    break;
case(6): // angle
    if (neg_flag)
    {
        gps_angle = 360 - UI_temp_calc;
        neg_flag = 0;
    }
    else
    {
        gps_angle = UI_temp_calc;
    }

    UI_temp_calc = 0;
    gps_num = 4;
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter month number: ");
    break;
case(7): // month
    if (neg_flag)
    {
        gps_month = UI_temp_calc;
        neg_flag = 0;
    }
    else
    {
        gps_month = UI_temp_calc;
    }

    UI_temp_calc = 0;
    gps_num = 5;
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter day number: ");
    break;
case(8): // day
    if (neg_flag)
    {
        gps_day = UI_temp_calc;
        neg_flag = 0;
    }
    else
    {
        gps_day = UI_temp_calc;
    }

    UI_temp_calc = 0;
    gps_num = 6;
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter year number: ");
    break;
case(9): // year
    if (neg_flag)
    {
        gps_year = UI_temp_calc;
        neg_flag = 0;
    }
    else
    {

```

```

        gps_year = UI_temp_calc;
    }

    UI_temp_calc = 0;
    gps_num = 7;
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter hour of the day (0-23): ");
    break;
case(10): // hours
    if (neg_flag)
    {
        gps_hours = UI_temp_calc;
        neg_flag = 0;
    }
    else
    {
        gps_hours = UI_temp_calc;
    }

    UI_temp_calc = 0;
    gps_num = 8;
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter the current time (minutes): ");
    break;
case(11): // minutes
    if (neg_flag)
    {
        gps_minutes = UI_temp_calc;
        neg_flag = 0;
    }
    else
    {
        gps_minutes = UI_temp_calc;
    }

    UI_temp_calc = 0;
    gps_num = 0;
    UI_state = 3;
    done_num_flag = 0;
    tower_num = 1;
    sun_az = find_azimuth(gps_long, gps_lat, gps_month, gps_day,
gps_hours, gps_minutes, 0, gps_angle);
    printf("GPS calculated sun_az: %f \r\n", sun_az);
    sun_alt = find_altitude(gps_long, gps_lat, gps_month, gps_day,
gps_hours, gps_minutes, 0);
    printf("GPS calculated sun_alt: %f \r\n", sun_alt);
    sun_az = az_1_a2d(sun_az + AZ_ZERO_1);
    sun_alt = alt_1_a2d(sun_alt + 15);
    printf("GPS input done. \r\n");
    printf("Enter the depth from the camera to the tower: ");
    break;
case(12): // depth of tower
    if (neg_flag)
    {
        tower_1_y = UI_temp_calc;
        neg_flag = 0;
    }
    else
    {
        tower_1_y = UI_temp_calc;
    }

    L = tower_1_y;
    UI_temp_calc = 0;
    tower_num = 2;
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter the horizontal offset of the tower: ");
    break;

```

```

case(13): // horizontal offset of tower
    if (neg_flag)
    {
        tower_1_x = -1*UI_temp_calc;
        neg_flag = 0;
    }
    else
    {
        tower_1_x = UI_temp_calc;
    }

    UI_temp_calc = 0;
    tower_num = 3;
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter the height of the tower: ");
    break;
case(14): // height of tower
    if (neg_flag)
    {
        tower_1_z = UI_temp_calc - 14;
        neg_flag = 0;
    }
    else
    {
        tower_1_z = UI_temp_calc - 14;
    }
    UI_temp_calc = 0;

    if (f_fabs(tower_1_x) < f_fabs(tower_1_y))
    {
        tower_az = az_1_a2d(f_atan2(tower_1_x,tower_1_y)*R2D + 90);
    }
    else
    {
        tower_az = az_1_a2d(90 - f_atan2(tower_1_y,tower_1_x)*R2D +
90);
    }
    if (f_fabs(tower_1_z) <
f_fabs(isqrt(f_power(tower_1_x,2)+f_power(tower_1_y,2))))
    {
        tower_alt =
alt_1_a2d(f_atan2(tower_1_z,isqrt(f_power(tower_1_x,2)+f_power(tower_1_y,2)))*R2D);
    }
    else
    {
        tower_alt = alt_1_a2d(90 -
f_atan2(isqrt(f_power(tower_1_x,2)+f_power(tower_1_y,2)),tower_1_z)*R2D);
    }
    printf("tower_1_x: %f \r\n tower_1_y: %f\r\n tower_1_z:
%f\r\n",tower_1_x,tower_1_y,tower_1_z);
    printf("tower_az: %f \r\n",tower_az);
    printf("tower_alt: %f \r\n",tower_alt);

    tower_num = 0;
    done_num_flag = 0;
    if (option_flag == 2)
    {
        printf("Enter the depth from the camera to the light
source: ");

        sun_num = 1;
        UI_state = 3;
    }
    else if (option_flag == 1)
    {
        UI_state = 18; // dead state
    }
    break;
case(15): // depth of sun
    if (neg_flag)
    {

```

```

        sun_1_y = -1*UI_temp_calc;
        neg_flag = 0;
    }
    else
    {
        sun_1_y = UI_temp_calc;
    }

    L_sun = sun_1_y;
    UI_temp_calc = 0;
    sun_num = 2;
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter the horizontal offset of the light source: ");
    break;
case(16): // horizontal offset of sun
    if (neg_flag)
    {
        sun_1_x = -1*UI_temp_calc;
        neg_flag = 0;
    }
    else
    {
        sun_1_x = UI_temp_calc;
    }

    UI_temp_calc = 0;
    sun_num = 3;
    UI_state = 3;
    done_num_flag = 0;
    printf("Enter the height of the light source: ");
    break;
case(17): // height of sun
    if (neg_flag)
    {
        sun_1_z = UI_temp_calc - 14;
        neg_flag = 0;
    }
    else
    {
        sun_1_z = UI_temp_calc - 14;
    }

    UI_temp_calc = 0;
    sun_num = 0;
    UI_state = 18;
    done_num_flag = 0;

    if (f_fabs(sun_1_x) < f_fabs(sun_1_y))
    {
        sun_az = az_1_a2d(f_atan2(sun_1_x,sun_1_y)*R2D + 90);
    }
    else
    {
        sun_az = az_1_a2d(90 - f_atan2(sun_1_y,sun_1_x)*R2D + 90);
    }
    if (f_fabs(sun_1_z) <
f_fabs(isqrt(f_power(sun_1_x,2)+f_power(sun_1_y,2))))
    {
        sun_alt =
alt_1_a2d(f_atan2(sun_1_z,isqrt(f_power(sun_1_x,2)+f_power(sun_1_y,2)))*R2D);
    }
    else
    {
        sun_alt = alt_1_a2d(90 -
f_atan2(isqrt(f_power(sun_1_x,2)+f_power(sun_1_y,2)),sun_1_z)*R2D);
    }
    printf("sun_alt: %f \r\n", sun_alt);
    printf("sun_1_x: %f \r\n sun_1_y: %f\r\n sun_1_z:
%f\r\n",sun_1_x,sun_1_y,sun_1_z);
    use_sun_pos_flag = 1;

```

```

        printf("UI done. \r\n");
        break;
    case(18): // dead state
        UI_done_flag = 1;
        break;
    }
}

// Servo Test function -- Used to test the Pololu mode of the servo controller
// range is 0 - 4000
void servo_test(unsigned int pos, unsigned char servo_number)
{
    unsigned char data1;
    unsigned char data2;

    pos += 1000; // 1.0 ms

    if (pos > current_pos+high_range)
    {
        current_pos += 1;
        pos_reached = 0;
    }
    else if (pos < current_pos-low_range)
    {
        current_pos -= 1;
        pos_reached = 0;
    }
    else if (pos > current_pos)
    {
        current_pos++;
        pos_reached = 0;
    }
    else if (pos < current_pos)
    {
        current_pos--;
        pos_reached = 0;
    }
    else if (pos == current_pos)
    {
        pos_reached = 1;
    }

    data1 = current_pos >> 7;
    if (data1 > 128)
    {
        data1 -= 128;
    }
    data2 = current_pos;
    if (data2 > 128)
    {
        data2 -= 128;
    }

    uart1SendChar(start_byte);
    uart1SendChar(device_ID);
    uart1SendChar(command);
    uart1SendChar(servo_number);
    uart1SendChar(data1);
    uart1SendChar(data2);

    return;
}

// Estimates the altitude of the Sun based on the time and location of the heliostat
float find_altitude (float longitude, float latitude, unsigned short month, unsigned
short day, unsigned short hour, unsigned short minutes, unsigned short seconds)
{
    unsigned short day_actual; // day of the year on 1-365 scale
    float g; // fractional year in radians

```

```

float eqtime;           // equation of time in minutes
float decl;             // solar declination angle in radians
float time_offset;      // time offset based on timezone in minutes
float tst;              // true solar time in minutes
float ha;               // solar hour angle
float zen_angle;        // solar zenith angle
float altitude;         // solar altitude
float pi = 3.14159;

switch (month)
{
    case(1):
        day_actual = day;
        break;
    case(2):
        day_actual = day + jan;
        break;
    case(3):
        day_actual = day + jan + feb;
        break;
    case(4):
        day_actual = day + jan + feb + mar;
        break;
    case(5):
        day_actual = day + jan + feb + mar + apr;
        break;
    case(6):
        day_actual = day + jan + feb + mar + apr + may;
        break;
    case(7):
        day_actual = day + jan + feb + mar + apr + may + jun;
        break;
    case(8):
        day_actual = day + jan + feb + mar + apr + may + jun + jul;
        break;
    case(9):
        day_actual = day + jan + feb + mar + apr + may + jun + jul + aug;
        break;
    case(10):
        day_actual = day + jan + feb + mar + apr + may + jun + jul + aug +
sep;
        break;
    case(11):
        day_actual = day + jan + feb + mar + apr + may + jun + jul + aug +
sep + octo;
        break;
    case(12):
        day_actual = day + jan + feb + mar + apr + may + jun + jul + aug +
sep + octo + nov;
        break;
}

g = 2*pi/365*(day_actual - 1 + (hour-12)/24.0);

eqtime = 229.18*(.000075 + 0.001868*f_cos(g) - .032077*f_sin(g) -
.014615*f_cos(2*g) - .040849*f_sin(2*g));

decl = .006918 - .399912*f_cos(g) + .070257*f_sin(g) - .006758*f_cos(2*g) +
.000907*f_sin(2*g) - .002697*f_cos(3*g) + .00148*f_sin(3*g);

time_offset = eqtime - 4*longitude + 60*timezone;

tst = hour*60 + minutes + seconds/60 + time_offset;

ha = (tst/4) - 180;           // in degrees
ha *= D2R;                   // in radians

zen_angle = f_acos(f_sin(latitude*D2R)*f_sin(decl) +
f_cos(latitude*D2R)*f_cos(decl)*f_cos(ha)); // in radians

altitude = 90.0 - zen_angle*R2D; // in degrees

```



```

        return (altitude);
    } // function end

// Estimates the azimuth angle of the sun based on the time and location of the heliostat
float find_azimuth (float longitude, float latitude, unsigned short month, unsigned short
day, unsigned short hour, unsigned short minutes, unsigned short seconds, float angle)
{
    unsigned short day_actual; // day of the year on 1-365 scale
    float g; // fractional year in radians
    float eqtime; // equation of time in minutes
    float decl; // solar declination angle in radians
    float time_offset; // time offset based on timezone in minutes
    float tst; // true solar time in minutes
    float ha; // solar hour angle
    float zen_angle; // solar zenith angle
    float azimuth; // solar azimuth
    float pi = 3.14159;

    switch (month)
    {
        case(1):
            day_actual = day;
            break;
        case(2):
            day_actual = day + jan;
            break;
        case(3):
            day_actual = day + jan + feb;
            break;
        case(4):
            day_actual = day + jan + feb + mar;
            break;
        case(5):
            day_actual = day + jan + feb + mar + apr;
            break;
        case(6):
            day_actual = day + jan + feb + mar + apr + may;
            break;
        case(7):
            day_actual = day + jan + feb + mar + apr + may + jun;
            break;
        case(8):
            day_actual = day + jan + feb + mar + apr + may + jun + jul;
            break;
        case(9):
            day_actual = day + jan + feb + mar + apr + may + jun + jul + aug;
            break;
        case(10):
            day_actual = day + jan + feb + mar + apr + may + jun + jul + aug +
sep;
            break;
        case(11):
            day_actual = day + jan + feb + mar + apr + may + jun + jul + aug +
sep + octo;
            break;
        case(12):
            day_actual = day + jan + feb + mar + apr + may + jun + jul + aug +
sep + octo + nov;
            break;
    }

    g = pi/180*(day_actual - 1 + (hour-12)/24);

    eqtime = 229.18*(.000075 + 0.001868*f_cos(g) - .032077*f_sin(g) -
.014615*f_cos(2*g) - .040849*f_sin(2*g));

```

```

    decl = .006918 - .399912*f_cos(g) + .070257*f_sin(g) - .006758*f_cos(2*g) +
    .000907*f_sin(2*g) - .002697*f_cos(3*g) + .00148*f_sin(3*g);

    time_offset = eqtime - 4*longitude + 60*timezone;

    tst = hour*60 + minutes + seconds/60 + time_offset;

    ha = (tst/4) - 180;                // in degrees
    ha *= pi/180;                      // in radians

    zen_angle = f_acos(f_sin(latitude*pi/180)*f_sin(decl) +
    f_cos(latitude*pi/180)*f_cos(decl)*f_cos(ha)); // in radians

    azimuth = 360 - 180/pi*f_acos(-(f_sin(latitude*pi/180)*f_cos(zen_angle) -
    f_sin(decl))/(f_cos(latitude*pi/180)*f_sin(zen_angle))); // in degrees, double check 360

    if (hour < 12)
    {
        azimuth = 360 - azimuth;
    }

    return(azimuth - angle);
} // function end

void calib(void)
{
    unsigned char ch;
    camera_reset(320);
    delayMS(2000);
    real_sun = 1;
    AWB_init();
    set_color_bin(RED);
    calib_done = 0;
    move_mirror_num = 0;
    while(calib_done == 0)
    {
        ch = getch();
        printf("%c\r\n",ch);
        switch(ch)
        {
            case ('1'):
                move_mirror_num = 1;
                break;
            case ('2'):
                move_mirror_num = 2;
                break;
            case ('3'):
                move_mirror_num = 3;
                break;
            case ('4'):
                move_mirror_num = 4;
                break;
            case ('e'):
                printf("Exiting Calibration.\r\n");
                calib_done = 1;
                break;
            case ('w'):
                switch(move_mirror_num)
                {
                    case (1):
                        desired_alt_camera += TEST_STEP;
                        while (!alt1flag)
                        {
                            move_servo(ALT1,desired_alt_camera);
                        }
                        alt1flag = 0;
                        printf("alt 1: %d \r\n",desired_alt_camera);
                        test_temp = alt_1_d2a(desired_alt_camera);
                        printf("alt 1(deg) = %f \r\n",test_temp);
                        break;
                }
        }
    }
}

```

```

        case (2):
            desired_alt_mirror_2 += TEST_STEP;
            while (!alt2flag)
            {
                move_servo(ALT2,desired_alt_mirror_2);
            }
            alt2flag = 0;
            printf("alt 2: %d
\r\n",desired_alt_mirror_2);

            test_temp = alt_2_d2a(desired_alt_mirror_2);
            printf("alt 2(deg) = %f \r\n",test_temp);
            break;
        case (3):
            desired_alt_mirror_3 += TEST_STEP;
            while (!alt3flag)
            {
                move_servo(ALT3,desired_alt_mirror_3);
            }
            alt3flag = 0;
            printf("alt 3: %d
\r\n",desired_alt_mirror_3);

            test_temp = alt_3_d2a(desired_alt_mirror_3);
            printf("alt 3(deg) = %f \r\n",test_temp);
            break;
        case (4):
            desired_alt_mirror_4 += TEST_STEP;
            while (!alt4flag)
            {
                move_servo(ALT4,desired_alt_mirror_4);
            }
            alt4flag = 0;
            printf("alt 4: %d
\r\n",desired_alt_mirror_4);

            test_temp = alt_4_d2a(desired_alt_mirror_4);
            printf("alt 4(deg) = %f \r\n",test_temp);
            break;
        default:
            printf("No mirror selected. \r\n");
            printf("1 - Mirror 1 \r\n2 - Mirror 2\r\n3 -
Mirror 3\r\n4 - Mirror 4\r\n");
            break;
    }
    break;
case ('a'):
    switch(move_mirror_num)
    {
        case (1):
            desired_az_camera -= TEST_STEP;
            while (!az1flag)
            {
                move_servo(AZ1,desired_az_camera);
            }
            az1flag = 0;
            printf("az 1: %d \r\n",desired_az_camera);
            test_temp = az_1_d2a(desired_az_camera) -
            printf("az 1 (deg): %f \r\n",test_temp);
            break;
        case (2):
            desired_az_mirror_2 -= TEST_STEP;
            while (!az2flag)
            {
                move_servo(AZ2,desired_az_mirror_2);
            }
            az2flag = 0;
            printf("az 2: %d \r\n",desired_az_mirror_2);
            test_temp = az_2_d2a(desired_az_mirror_2) -

```

```

        printf("az 2 (deg): %f \r\n",test_temp);
        break;
    case (3):
        desired_az_mirror_3 -= TEST_STEP;
        while (!az3flag)
        {
            move_servo(AZ3,desired_az_mirror_3);
        }
        az3flag = 0;
        printf("az 3: %d \r\n",desired_az_mirror_3);
        test_temp = az_3_d2a(desired_az_mirror_3) -

AZ_ZERO_3;

        printf("az 3 (deg): %f \r\n",test_temp);
        break;
    case (4):
        desired_az_mirror_4 -= TEST_STEP;
        while (!az4flag)
        {
            move_servo(AZ4,desired_az_mirror_4);
        }
        az4flag = 0;
        printf("az 4: %d \r\n",desired_az_mirror_4);
        test_temp = az_4_d2a(desired_az_mirror_4) -

AZ_ZERO_4;

        printf("az 4 (deg): %f \r\n",test_temp);
        break;
    default:
        printf("No mirror selected. \r\n");
        printf("1 - Mirror 1 \r\n2 - Mirror 2\r\n3

- Mirror 3\r\n4 - Mirror 4\r\n");
        break;
    }
    break;
    case ('s'):
        switch(move_mirror_num)
        {
            case (1):
                desired_alt_camera -= TEST_STEP;
                while (!alt1flag)
                {
                    move_servo(ALT1,desired_alt_camera);
                }
                alt1flag = 0;
                printf("alt 1: %d \r\n",desired_alt_camera);
                test_temp = alt_1_d2a(desired_alt_camera);
                printf("alt 1(deg) = %f \r\n",test_temp);
                break;
            case (2):
                desired_alt_mirror_2 -= TEST_STEP;
                while (!alt2flag)
                {

                move_servo(ALT2,desired_alt_mirror_2);
                }
                alt2flag = 0;
                printf("alt 2: %d

\r\n",desired_alt_mirror_2);

                test_temp = alt_2_d2a(desired_alt_mirror_2);
                printf("alt 2(deg) = %f \r\n",test_temp);
                break;
            case (3):
                desired_alt_mirror_3 -= TEST_STEP;
                while (!alt3flag)
                {

                move_servo(ALT3,desired_alt_mirror_3);
                }
                alt3flag = 0;
                printf("alt 3: %d

\r\n",desired_alt_mirror_3);

                test_temp = alt_3_d2a(desired_alt_mirror_3);

```

```

        printf("alt 3(deg) = %f \r\n",test_temp);
        break;
    case (4):
        desired_alt_mirror_4 -= TEST_STEP;
        while (!alt4flag)
        {
            move_servo(ALT4,desired_alt_mirror_4);
        }
        alt4flag = 0;
        printf("alt 4: %d
\r\n",desired_alt_mirror_4);

        test_temp = alt_4_d2a(desired_alt_mirror_4);
        printf("alt 4(deg) = %f \r\n",test_temp);
        break;
    default:
        printf("No mirror selected. \r\n");
        printf("1 - Mirror 1 \r\n2 - Mirror 2\r\n3 -
Mirror 3\r\n4 - Mirror 4\r\n");
        break;
    }
    break;
case ('d'):
    switch(move_mirror_num)
    {
        case (1):
            desired_az_camera += TEST_STEP;
            while (!az1flag)
            {
                move_servo(AZ1,desired_az_camera);
            }
            az1flag = 0;
            printf("az 1: %d \r\n",desired_az_camera);
            test_temp = az_1_d2a(desired_az_camera) -

AZ_ZERO_1;

            printf("az 1 (deg): %f \r\n",test_temp);
            break;
        case (2):
            desired_az_mirror_2 += TEST_STEP;
            while (!az2flag)
            {
                move_servo(AZ2,desired_az_mirror_2);
            }
            az2flag = 0;
            printf("az 2: %d \r\n",desired_az_mirror_2);
            test_temp = az_2_d2a(desired_az_mirror_2) -

AZ_ZERO_2;

            printf("az 2 (deg): %f \r\n",test_temp);
            break;
        case (3):
            desired_az_mirror_3 += TEST_STEP;
            while (!az3flag)
            {
                move_servo(AZ3,desired_az_mirror_3);
            }
            az3flag = 0;
            printf("az 3: %d \r\n",desired_az_mirror_3);
            test_temp = az_3_d2a(desired_az_mirror_3) -

AZ_ZERO_3;

            printf("az 3 (deg): %f \r\n",test_temp);
            break;
        case (4):
            desired_az_mirror_4 += TEST_STEP;
            while (!az4flag)
            {
                move_servo(AZ4,desired_az_mirror_4);
            }
            az4flag = 0;
            printf("az 4: %d \r\n",desired_az_mirror_4);
            test_temp = az_4_d2a(desired_az_mirror_4) -

AZ_ZERO_4;

```

```

        printf("az 4 (deg): %f \r\n", test_temp);
        break;
    default:
        printf("No mirror selected. \r\n");
        printf("1 - Mirror 1 \r\n2 - Mirror 2\r\n3
- Mirror 3\r\n4 - Mirror 4\r\n");
        break;
    }
    break;
}
}
return;
}

// Servo Calibration Functions
float az_1_d2a(float result)
{
    result = 0.0000000001*f_power(result,3) + 3*0.0000001*f_power(result,2) +
0.0485*result - 14.53;
    return result;
}
float az_1_a2d(float result)
{
    result = -2*0.00001*f_power(result,3) + 0.0004*f_power(result,2) + 20.256*result +
302.45;
    return result;
}
float alt_1_d2a(float result)
{
    result = -5*0.0000000001*f_power(result,3) + 3*0.000001*f_power(result,2) +
0.0476*result + 2.0811;
    return result;
}
float alt_1_a2d(float result)
{
    result = 6*0.00001*f_power(result,3) - 0.0172*f_power(result,2) + 20.452*result -
32.551;
    return result;
}

float az_2_d2a(float result)
{
    result = -4*0.0000000001*f_power(result,3) + 2*0.000001*f_power(result,2) +
0.0508*result - 14.821;
    return result;
}
float az_2_a2d(float result)
{
    result = 5*0.00001*f_power(result,3) - 0.0101*f_power(result,2) + 19.203*result +
289.81;
    return result;
}
float alt_2_d2a(float result)
{
    result = -4*0.0000000001*f_power(result,3) - 2*0.000001*f_power(result,2) +
0.0536*result + 0.6327;
    return result;
}
float alt_2_a2d(float result)
{
    result = -1*0.00001*f_power(result,3) + 0.006*f_power(result,2) + 18.238*result -
4.4679;
    return result;
}

float az_3_d2a(float result)
{
    result = -7*0.0000000001*f_power(result,3) + 4*0.000001*f_power(result,2) +
0.0445*result - 12.988;
    return result;
}

```

```

float az_3_a2d(float result)
{
    result = 9*0.00001*f_power(result,3) - 0.0232*f_power(result,2) + 21.015*result +
287.48;
    return result;
}
float alt_3_d2a(float result)
{
    result = -5*0.0000000001*f_power(result,3) + 2*0.000001*f_power(result,2) +
0.052*result + 2.5201;
    return result;
}
float alt_3_a2d(float result)
{
    result = 5*0.00001*f_power(result,3) - 0.0103*f_power(result,2) + 18.948*result -
42.23;
    return result;
}

float az_4_d2a(float result)
{
    result = 1*0.00000000001*f_power(result,3) + 1*0.0000001*f_power(result,2) +
0.0513*result - 16.969;
    return result;
}
float az_4_a2d(float result)
{
    result = -2*0.00001*f_power(result,3) + 0.0031*f_power(result,2) + 19.182*result +
333.01;
    return result;
}
float alt_4_d2a(float result)
{
    result = -3*0.0000000001*f_power(result,3) + 1*0.000001*f_power(result,2) +
0.0509*result + 0.5091;
    return result;
}
float alt_4_a2d(float result)
{
    result = 4*0.00001*f_power(result,3) - 0.0088*f_power(result,2) + 19.479*result -
5.4594;
    return result;
}

float az_calib(float result)
{
    return result;
}

float alt_calib(float result)
{
    return result;
}

float az_calib_sun(float result)
{
    return result;
}

// Math Functions -- All angles are in radians
// Taylor Series approx. for sine
float f_sin(float a)
{
    int i;
    int N; //update to the Nth order of Taylor series
    N=20;
    int sign;
    sign=1;
    float result;
    result = 0;

```

```

        for (i = 1; i < N; i += 2)
        {
            result += f_power(a,i)/f_factorial(i)*sign;
            sign *= -1;
        }

        return result;
    }

// Taylor Series approx. for cosine
float f_cos(float a)
{
    int i;
    int N;
    N = 20;
    int sign;
    sign = -1;
    float result;
    result = 1;

    for (i = 2; i < N; i += 2)
    {
        result += f_power(a,i)/f_factorial(i)*sign;
        sign *= -1;
    }
    return result;
}

float f_asin(float a)
{
    float result;
    int i;
    result = 0;

    if (f_fabs(a) > 1.0)
    {
        return 0;
    }
    else
    {
        for(i = 0; i < 10; i++)
        {
            result +=
f_factorial(2*i)/(f_power(4,i)*f_power(f_factorial(i),2)*(2*i+1))*f_power(a,2*i+1);
        }

        return result;
    }
}

// Arccosine
float f_acos(float a)
{
    float pi=3.1416;
    float result = 0;
    result = pi/2.0 - f_asin(a);
    return (result);
}

// Arctan
float f_atan(float a){
    float result;
    result = 0;
    int i;
    int sign;
    sign = 1;
    for (i = 1; i < 100; i += 2)
    {
        result += f_power(a,i)/i*sign;
        sign *= -1;
    }
}

```



```

    }

    return result;
}

// Atan2
float f_atan2(float y, float x)
{
    float result;
    float pi=3.1416;

    if (x > 0)
        result = f_atan(y/x);
    else if (y >= 0 && x < 0)
        result = pi + f_atan(y/x);
    else if (y < 0 && x < 0)
        result = -pi + f_atan(y/x);
    else if (y > 0 && x == 0)
        result = pi/2;
    else if (y < 0 && x == 0)
        result = -pi/2;
    else
        result = 1.0e8;

    return result;
}

// Sqrt
float f_sqrt(float a)
{
    float a0 = a;
    float c0 = a - 1;
    int i;

    for (i = 0; i < 20; i++)
    {
        a0 = a0 - a0*c0/2.0;
        c0 = c0*c0*(c0-3)/4.0;
    }

    return a0;
}

// Factorial
unsigned int f_factorial(unsigned int N){

    unsigned int result = 1;
    unsigned int i;

    if (N == 0)
        result = 1;
    else if (N >= 1)
    {
        for(i = 1; i < N+1; i++)
            result *= i;
    }
    else
        result = 0;

    return result;
}

// Power
float f_power(float x, int N){

    float result;
    int i;
    result = 1;

    if (N == 0)
        return 1;

```

```

        else if (N > 0) {
            for(i = 0; i < N; i++)
                result *= x;
            return result;
        }
        else
            return 0;
    }

// Absolute Value
float f_fabs(float x)
{
    float lb = -1e-6;
    float rb = 1e-6;

    if (x > rb)
        return x;
    else if(x<lb)
        return (-x);
    else
        return 0;
}

// Tangent
float f_tan(float x)
{
    float result;
    result = f_sin(x)/f_cos(x);
    return result;
}

```