UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Automatic Aligning Free Space Communication Platform

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science

in

Electrical Engineering (Photonics)

by

John Michael Andrews

Committee in charge:

Professor Joseph Ford, Chair
Professor George Papen
Professor Shayan Mookherjea

2008

The Thesis of John Michael Andrews is approved and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2008

## TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT OF THE THESIS

Automatic Aligning Free Space Communication Platform

by

John Michael Andrews

Master of Science in Electrical Engineering (Photonics)

University of California, San Diego, 2008

Professor Joseph Ford, Chair

Recent developments in ultra-dense optical components coupled with micro-scale electro-mechanical systems have led to the ability to create a new type of optical device: the free-space self-aligning optical transmitter/receiver. With automatic search and alignment, high-speed data or power transfer, and low-power operation, this new tool is poised to become an important tool in many systems.  Utilizing MEMS technology and high speed electronic control and sensing, one half of this device allows for independent beam control, automatic beam scanning and steering, as well as incoming beam detection and positioning.  Pairing this host module up with a client module outfitted with a new high-speed optical modulator (also developed at UCSD) and a classic corner cube reflector, yields a unique blend of technologies that allows for secure high-speed point-to-point communications or even remote sensing and power transfer systems.

# INTRODUCTION

Recent developments in ultra-dense optical components coupled with micro-scale electro-mechanical systems have led to the ability to create a new type of optical device: the free-space self-aligning optical transmitter/receiver. With automatic search and alignment, high-speed data or power transfer, and low-power operation, this new tool is poised to become an important tool in many systems. Utilizing MEMS technology and high speed electronic control and sensing, one half of this device allows for independent beam control, automatic beam scanning and steering, as well as incoming beam detection and positioning. Pairing this host module up with a client module outfitted with a new high-speed optical modulator (developed at UCSD) and a classic corner cube reflector, yields a unique blend of technologies that allows for secure high-speed point-to-point communications or even remote sensing and power transfer systems.

In the last decade, secure point-to-point communications has become a hot commodity. Radio systems have the distinct disadvantage of being omnidirectional, such that any nearby parties can listen in with little fear of detection. As such, optical communication systems have sprung to accommodate this security need. Companies such as LightPointe (www.lightpointe.com) provide secure full 100+ Mbps point-to-point optical communication systems. However, such systems may have a prohibitively high cost, often due to the need for precise alignment between transceivers. Also, such systems are limited to a single pairing of devices, and as such, cannot accommodate new clients into the infrastructure. An optical multipoint communication system using one or more simple client transceivers and coupled with a host transceiver would be greatly effective at fulfilling the secure transmission need as well as the adaptability of a dynamic network. Add in automatic searching and client discovery and many clients can be used simultaneously with a single host. For example, in a typical office application, the host system could be placed in areas where data communication requirements are high, such as in private conference rooms or secure facilities. Small business electronics, such as laptops, cell phones, and PDAs, could be embedded with the small client transceiver, which would allow communications with the local intranet and out to the internet, if desired. The host could also be used in areas where existing wireless transmissions are too insecure for data transmission, as the system not only has point-to-point connections, but also automatic detection of connection loss as well as detection of any third-party eavesdroppers.

Another venue for this technology, possibly even more useful than communication, is wireless power transmission. This too, has been a dream for engineers, and is as yet unsolved for consumer applications. Imagine leaving a cell phone or PDA on a table in a room containing the wireless transmitter, and having it trickle charge the device's battery without having to use any plugs or wires. Utilizing the automatic connection loss notification also prevents a high powered optical beam from harming anything passing through it, as the laser power could be reduced to safe levels within microseconds.

One key element in this system is a newly developed modulating retro-reflector, designed at UCSD by Trevor Chan and Dr. Joseph Ford. This modulator is a newly developed MEMS diffractive element modulator that directly transports an analog voltage signal onto an incident laser beam. The modulator produces a high contrast signal using low power consumption. By combining this modulator with a pair of non-modulating mirrors, it can be converted into a retroreflecting corner cube reflector. This allows the reflected beam to be directed back to the source system without any alignment issues. One additional advantage is that this modulator utilizes standard lithographic fabrication techniques, making the per-unit cost very low, thus making the client side system cheap and easy to manufacture.

This project aims to implement this new communication or power delivery paradigm, by designing and building both the complex host transceiver and the simple client transceiver, as described above. The project consists of 4 separate stages of development and integration. The first two stages involve hardware selection and system design. The third stage mainly involves software command and control development. The fourth stage adds the reactive feedback layer and data transfer subsystems.

## OVERVIEW

The complex host side consists of a high speed laser tracking system utilizing electronically steerable MEMS mirrors, as well as a suite of sensors dedicated to detecting the returning signals. Ideally, the simple client side consists of a corner cube reflector with a built in high speed modulator and detector. However, in this first iteration of the project, the modulating corner cube reflector is solely a transmitter, and does not have any signal detection abilities. Its purpose is to add data to the incoming beam, and will automatically direct the beam back to the host detector.

The system consists of multiple hardware and software components and subsystems. The hardware components consist of electronic devices and optical elements. The software components consist of back-end communication protocols, search algorithms, and tracking routines. All of these subsystems work together to create the host transceiver.

The software subsystems take the high level control of the entire system, providing advanced logic and control to the hardware subsystems. The major components of the high level software logic are the search routine, which locates and locks on to the remote target, and the track routine, which maintains a lock on the target. The high level software communicates to the hardware using a low level interface software layer. This interface translates high level commands, such as a “move” or “get power” command into the hardware signals.

The hardware subsystems receive these commands and process them as needed. Commands sent to the optical subsystem pass through the TALP4020 board, which contains the hardware logic necessary to decode and process them. In the case of mirror commands, they are then routed through a DAC, and then to the mirror.  For commands sent to the sensor subsystem, a data acquisition card is used.  Through this card, the pc acquires the voltages that the sensors output.

To begin building this complicated system, the project was divided into four stages of work.  The initial stage involved verifying functionality of all the hardware. This stage involved simple signal detection using oscilloscopes as well as basic code to control mirror pointing.  The second stage incorporated more of the hardware into the system, including extra mirrors, lasers, and other devices.  The third stage added the modulator to the system, and converted the target from a power detector to a mirror.  The fourth and final stage replaced the target mirror with the modulating reflector. Thus an adequate bench model that implemented all the desired features of the original design was created.

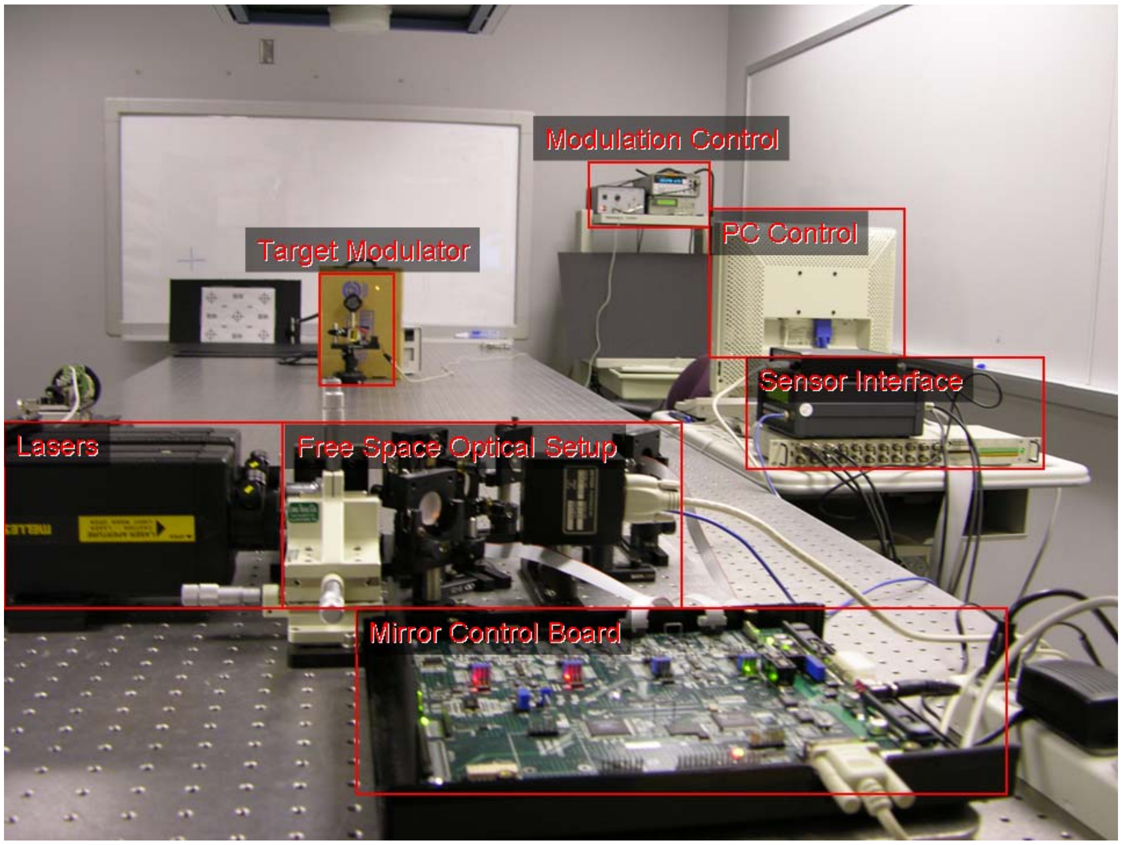


**Figure 1 - System Lab Setup**

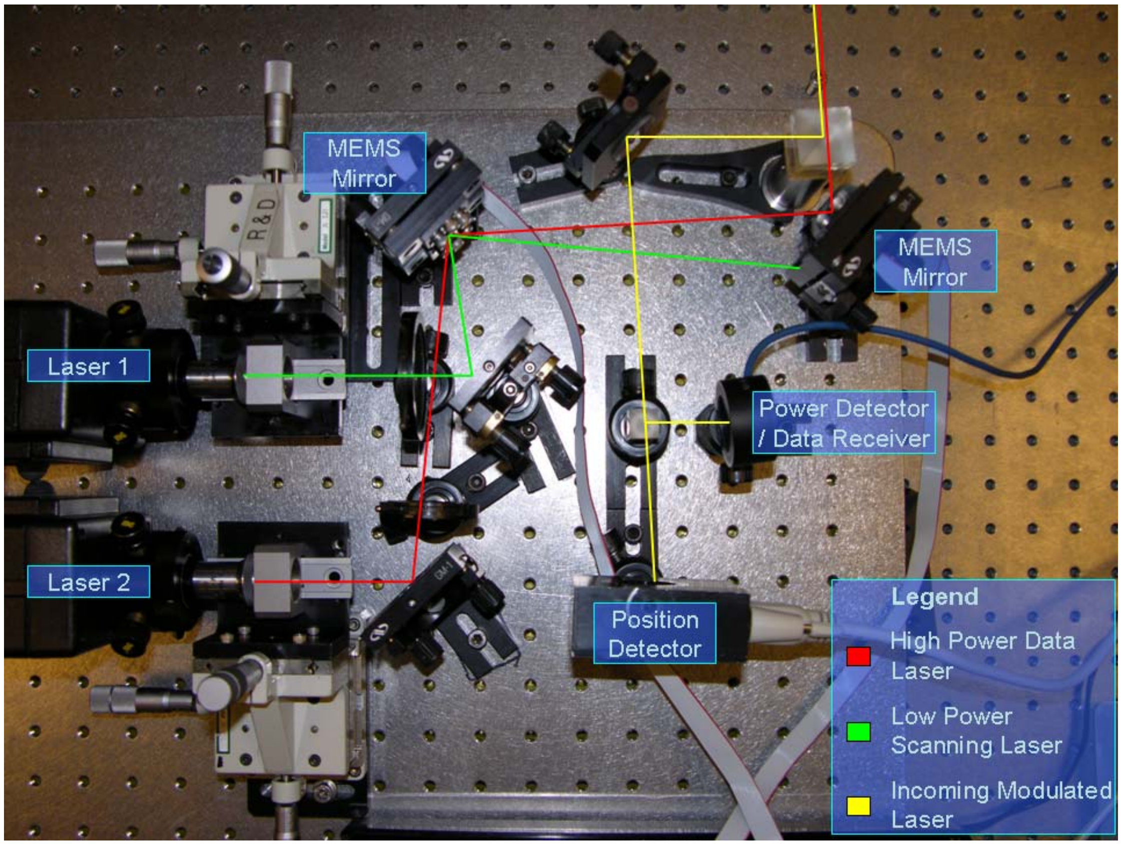

MEMS Mirror
MEMS Mirror
Laser 1
Power Detector / Data Receiver
Laser 2
Position Detector
Legend
🟥 High Power Data Laser
🟩 Low Power Scanning Laser
🟨 Incoming Modulated Laser

**Figure 2 - Free Space Optical Setup**

# SECTION 1 –DESIGNS AND IMPLEMENTATION

## Stage I – MEMS Mirror Control

The goal of the first stage of the project was to get the MEMS mirrors up and running. Initially, the hardware setup was limited to the mirror, controller board, and a laser. A simple test program was supplied with the mirror software, and was used test basic mirror functionality. Following the verification of the mirror hardware, scripts in Matlab were developed to provide functionality similar to the test program. Predefined sequences such as tilting the mirror to its extents, moving to preselected points, and basic row scanning were implemented with the basic code framework.

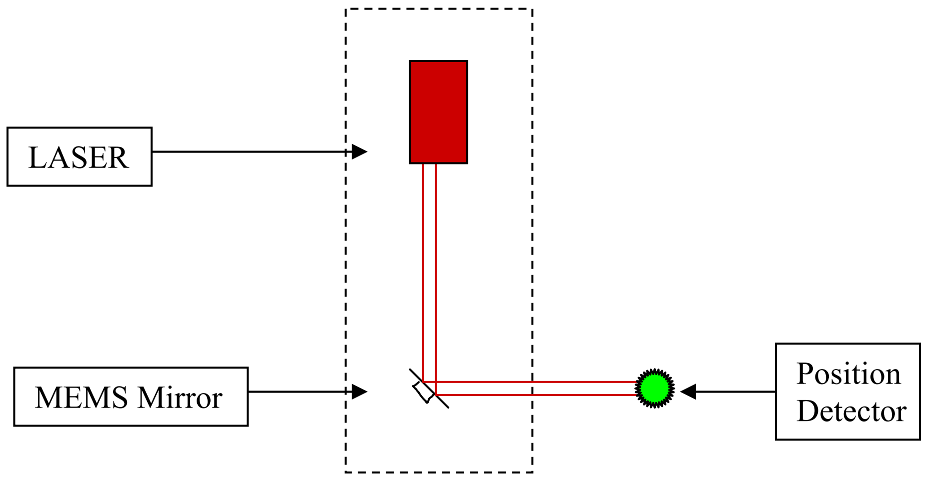


**Figure 3 – Stage 1 Diagram**
**MEMS Mirror Setup and Test**

Some basic system design ideas were developed and tested at this point. One such design goal was to provide a method of laser color switching. Ideas ranged from using a fiber optic switch to using a MEMS mirror. At this point, it was undecided whether to make a fiber optic system or to keep everything in free space. The fiber optic switch had a number of strong advantages associated with it, including fast switching time, simple control, and a single output point. One problem that came up was that there weren't any visible light fiber switches on hand at the lab. This, coupled with the high component insertion loss led to the adoption of the free space system design.

## Stage II – Laser Source Selection

The next stage of the project involved converting the fiber optic parts of the system to free space components, as well as adding a second laser to the system for the color switching. The second MEMS mirror was also added to the system in a configuration that allowed it to be used as the color switch.

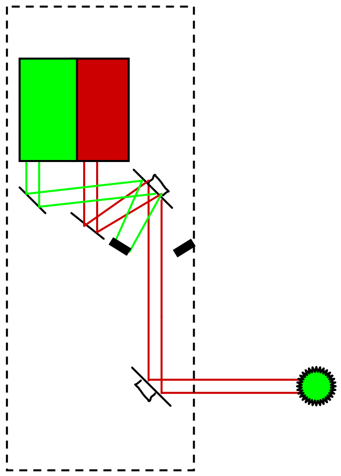


**Figure 4 – Stage 2 Diagram**
**Laser Switching Mechanism & Tracking Development**

These basic hardware changes necessitated the creation of a set of low level basic I/O code blocks, written in Matlab. These blocks are then used to create the higher level complex algorithms. Such high level functions included code for moving mirrors and reading sensors. These intermediate blocks were then used to create the high level search and seek algorithms.

One function this led to was the alignTool code. This consisted of a GUI that allows the user to move the color changing mirror to align one laser with the second mirror. It then saves the settings after the user has chosen an optimal location. [see Code Analysis section for more information]

The second mirror added a couple of other advantages to the overall system. Laser alignment with the scanning mirror could now be performed remotely by computer, instead of having to manually adjust the static mirrors by hand. Also, by removing the lossy fiber optic switching portion of the system, peak optical power could be maintained, which greatly assists in the detection of remote targets.

After these hardware changes were implemented, scanning and tracking algorithms were developed. In the beginning, a few different scanning algorithms were tested for speed and detection probability. Among these were a raster scan, circular scan, and "snake scan." The raster scan pattern was selected first for its simplicity of implementation. It scans a horizontal row of points, moves down a bit, and repeats. As scanning algorithms go, though, it is not very fast or reliable at detecting the target. However, for this project, getting a working model was a higher priority than getting a supper efficient prototype.

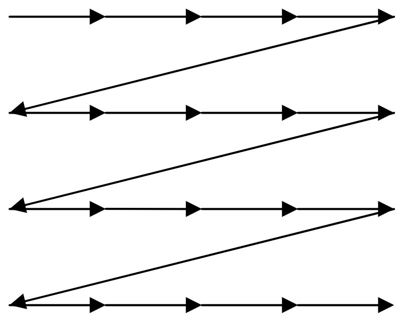


**Figure 5 - Raster Scan Pattern**

The circular scan pattern was implemented next because the thing we're searching for isn't generally going to be at the extreme edges of the field of view, so it seems ideal to concentrate our scanning time in the center of the field of view. This pattern had the advantage that it had a high scan point density around the center of the scan area. This scan was highly accurate in the laboratory setting, where the sensor was placed in the center of the scanning field, but has little use in a real world setting, where a target located near the outer extremities of the field had very little chance of being detected. The method of implementation basically set a radius, and scanned a fixed number of points in one revolution, then repeated this with a larger radius (see Fig. 6). This particular method had the disadvantage of leaving large gaps between points on the outer edge, possibly missing a reflector in the gap. It also had the disadvantage of needing some heavy post processing to reform the concentrically collected data points into their x-y coordinates. This idea was soon discarded, and a modification of the previous algorithm was attempted next.

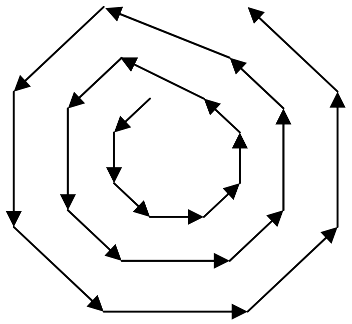


**Figure 6 - Circle Scan Pattern**

This led to a refinement of the raster scan into the “snake scan”. It is similar to the raster scan pattern, but reverses the horizontal scanning direction every other row. This pattern minimizes the distance the mirror moved between points. This improves scanning speed since the next point is physically closer to the previous point, as well as reducing any beam overshoot by not moving large distances. This pattern runs slightly faster than the raster scan, and has equal accuracy in locating targets. However, it adds a few complications to the coding. The data points are stored in the order they were scanned, and need to be reformatted into the correct grid pattern on each scan in order to avoid having every other line reversed.

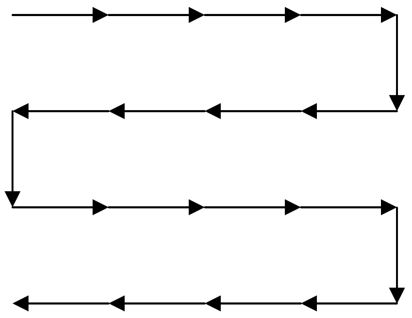


**Figure 7 - Snake Scan Pattern**

In the end, however, the simple raster scan was selected as the scanning algorithm of choice. Its simple implementation and high speed were key qualifiers in its selection. Scanning times with this algorithm generally run 20 to 60 second range, depending on scan point density, and provides adequate performance.

The tracking was also written at this point. It implements a simple negative feedback loop utilizing a position sensor as the input. The position information given by the sensor is used as the feedback parameter in the loop. In practice, the code reads the voltage returned from the sensor, maps it into sensor coordinates, then converts the sensor coordinates into physical mirror position coordinates. These final coordinates are multiplied by negative one and sent to the mirror as a relative position command. This quick implementation is very fast and efficient at tracking the target, however some care needs to be taken with the feedback gain multiplier used, or else the beam might jump off of the reflector, requiring reacquisition of the reflector.

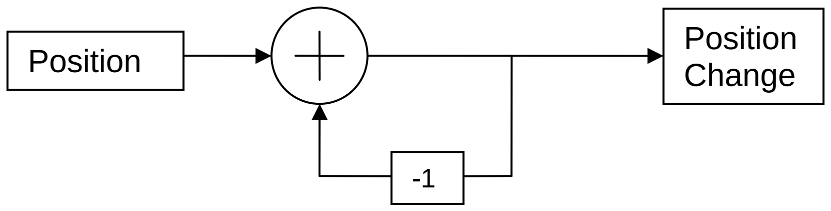


**Figure 8 – Position Feedback Loop**

## Stage III – Static Reflector and Detector

The third stage adds the corner cube reflector and the power detector to the system, alongside the existing position sensing detector. The addition of the power sensor is needed for operating the system in ambient light, as well as for detecting the modulated beam. The position sensor outputs vary significantly with changes in ambient light, a most undesirable effect in the system. A relative ambient light measurement can be obtained by the power sensor, thus allowing the system to gauge an appropriate correction factor. The power sensor also allows for rapid discovery of loss of target tracking.

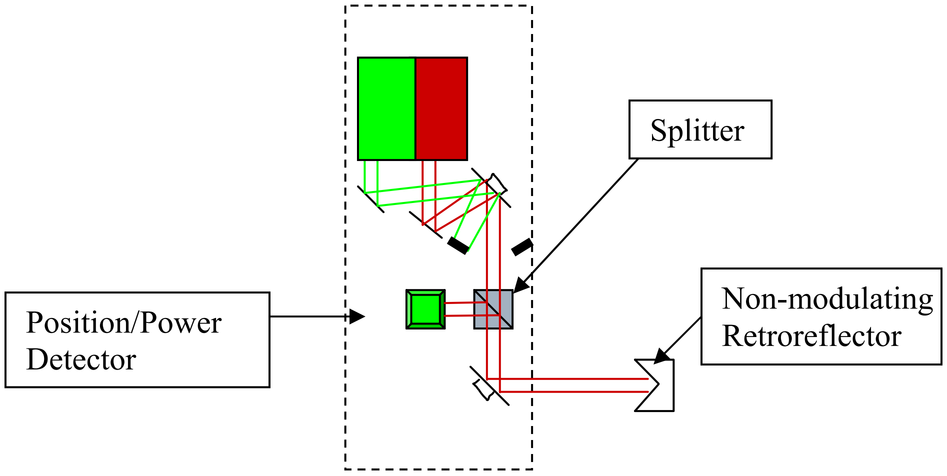


**Figure 9 – Stage 3 Diagram**
**Retroreflection and Detection and Tracking**

The addition of the passive corner cube reflector is significant since this allows for both the transmitter and receiver to be collocated. This more closely approximates our final system, where both transmitters and receivers in both local and remote locations are collocated.

Each of these additions requires minor code changes. The corner cube requires new code to allow for the mirroring of beam movements by the reflector. The power sensor code allows for extraction of the power level detected by the power sensor. One major advantage of the code blocks is the modular nature of the algorithm and its subfunctions. This means that most changes can be implemented quickly and easily by simply modifying the respective function. For example, when implementing the different scanning routines (described above), only one subfunction needed to be changed, and then, only the arrays of locations fed to the scanner. The remainder of the code was able to be reused, as is.

## Stage IV – Dynamic Reflector and Detector

The fourth and final stage of the project incorporates the remainder of the functionality. This includes replacing the passive corner cube reflector with the active modulating one, as well as incorporating “beam interrupt detection” functionality. The modulation is detected in software by analyzing the returned signal over a short time, and performing a FFT, and looking for the presence of the desired modulation frequency. The beam interrupt detection is implemented in the tracking routine by monitoring the returned signal for drops in power and changing the laser color (or power) accordingly.

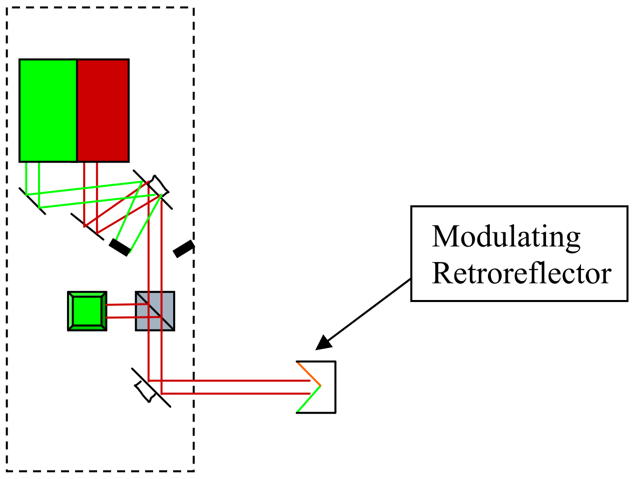


**Figure 10 – Stage 4 Diagram**
**Modulated Beam Detection and Tracking**

## System Level Testing

To verify the functionality of the system as a whole, a system level test has been developed. This system wide test consists of multiple parts, and generally a failure in one stage of testing causes failure in the following stages. The first test is to verify the communication system, so that commands can be given to the mirrors to perform seek and track operations. The second test is to exercise the search routine capabilities, so that the retromodulator can be located. The third test is to exercise the tracking subsystem, so that the datalink is kept open at all times. These tests validate the functionality of the system and will check for correct implementation and functionality of the algorithms and hardware components.

For the communication test, the two parts of the communication system are tested separately. The functionality of the serial port, TALP interfaces, and hardware memory modules are tested by attempting to read a block of memory on the mirror controller board. The TALP board returns a SUCCESS return code, along with the memory data, if the test is successful. The data acquisition system, including the data acquisition board, the sensor cabling, and the sensors themselves, are tested by attempting to read the power level of each attached sensor. Nonzero power levels will suffice to indicate a successful test. Success on both parts is required to perform the searching and tracking tests. Any further comprehensive hardware testing is manually performed at this stage of the project.

The search routine tests nearly every aspect of the system, including the functionality of the lasers, micro mirrors, sensors, and modulator. The test consists of placing two retro reflecting devices (one passive, one modulating) in the field of view of the scanning mirror, and verifying that the modulating one is found when searching for a modulating source, and the passive one is found when looking for a non-modulating one. The modulated reflector is driven by 50 kHz square wave. The system test is deemed a success when the scanning subsystem locks onto only the retro reflecting modulator.

The tracking routine test verifies the remainder of the hardware, as well as the tracking response time, and system tolerance to ambient light and optical interference.

The test starts with the beam locked onto the retro reflector, either automatically by success of the previous test, or manually by using debugging code. The first part of the test gauges the approximate velocity limit at which the target can be moving while the system maintains signal lock. In general, the test passes if the tracking system can maintain lock at any speed, no matter how slow. The second part tests the effect of ambient light on the tracking mechanism. Ambient light greatly influences the voltage readings of the position sensor, and this test measures the extent of this effect. The test passes if the system performs tracking adequately with room lights turned on. Success on this test, as well as the other two, verifies a completely functional system.

**Results of System Test**

The results of the above system level tests expose the particular strengths and weaknesses of the system as a whole. This section will discuss the results of the above system test on the final demonstration setup. The strengths will be noted, and the causes and possible solutions of each weakness will be discussed.

The communication system has been thoroughly tested to be rock solid. Since the beginning of the project, the mirror controller board has never misinterpreted a command sent to it, and no command has been malformed by board hardware or software bugs. This is a great boon to the system, as any errors produced at the hardware level would be practically indistinguishable from system level problems.

The search routine is fast and effective at locating either reflector. Though not a realtime scan, the algorithm scans the entire field of view in less than sixty seconds, with enough accuracy to find a small retroreflector. The search routine finds the correct sensor with about 90 percent accuracy, however the exact point on the reflector onto which the beam locks seems to vary from scan to scan. The method used to detect the modulated signal has worked admirably since its initial implementation, and in general, if the scanning beam passes over any part of the modulator, the detector finds the modulated signal. One complication in setting up the scan is to determine an optimum scanning point density. The scan point density varies with distance to the reflector, and with visible reflector area. A greater number of scanning points also allows for a much higher chance of detecting the modulator, however, the scanning time increases accordingly. Overall, however, the scanning subsystem works acceptably and passes the evaluation tests.

Below are the timing results of the SEEK routine, measured as the number of seconds to complete a full scan at the selected density. The rows vary the number of samples scanned in each direction, and the columns vary the number of data points that are averaged for each data sample. Generally, thirty to forty points in each direction are needed to find the reflector at a one to two meter distance, and more for longer distances. Limiting the points per sample to 200 seems to produce the fastest seek times.

| | **SEEK TIME** (sec) | | | | | | |
|---|---|---|---|---|---|---|---|
| | Data points per sample | | | | | | |
| Grid Size | 500 | 400 | 300 | 200 | 100 | 50 | 10 |
| 20x20 | 12.1 | 9.9 | 8.7 | 8.7 | 8.5 | 8.5 | 8.4 |
| 30x30 | 27.2 | 22.5 | 20.0 | 19.1 | 19.1 | 19.1 | 19.0 |
| 40x40 | 48.1 | 39.9 | 35.3 | 34.8 | 34.7 | 33.6 | 34.2 |
| 50x50 | 75.7 | 61.1 | 55.0 | 53.8 | 52.5 | 52.1 | 52.3 |

**Figure 11 -- Reflector Seek Time**

The tracking routine, however, does not fare as well on its tests. Tracking the passive reflector works nearly every time. However, due to a few inconsistencies on the modulating retro-reflector, tracking it proved to be difficult. Such problems included a smaller reflective area than the passive reflector, slightly misaligned mirror walls which caused the return beam to be misaligned, as well as a gap between two of the mirror walls which proved to be a dead spot. However, every effort was made to navigate around these problem areas, and tracking was successfully performed on most trial runs.

Results of the tracking speed test showed that the system was able to track a target moving about 10 mm/sec, whereas the tracking of the modulating reflector is about 25% of that, or 2 mm/sec. Other factors also contribute to these slow tracking speeds, including position sensor noise (mainly due to unshielded cabling), and the inability of the system design to lock on to the dead center of the device when utilizing an unpolarized light source. The beamsplitter will send a portion of the returned beam straight through to the micro mirror and cause the mirror's internal sensors to go haywire, making the system unable to lock on the dead center of the retro-reflector. Future revisions should utilize either polarized visible light with a polarized beamsplitter or simply standard IR communication laser wavelengths which the micro mirror is designed to handle. Despite these limitations, the system can still maintain a solid lock on the reflector in full ambient light, as well as perform the beam blockage color switching perfectly.

Below are the timing results of the TRACK routine, measured as feedback loop iterations per second. The columns represent the number of data points averaged per sample, as shown in the table above. The rows vary the sample rate of the data acquisition device, and as such, the maximum data rate of the client transceiver. As can be seen from the chart, there is really no performance based reason to not run the DAQ at its maximum rate of 66,000 samples per second, as the time per loop only decreases as sample rate increases, as is expected. Even at the maximum speed, which is almost 50Hz, the overwhelming amount of receiver noise present in the position detector causes the detected beam spot location to vary wildly.

**TRACKING UPDATES PER SECOND** (Hz)

| DAQ Sample Rate | Pts/sample | | | | | | |
|---|---|---|---|---|---|---|---|
| | 500 | 400 | 300 | 200 | 100 | 50 | 10 |
| 1kHz | 1.9 | 2.1 | 2.9 | 3.7 | 9.0 | 16.4 | 47.8 |
| 10kHz | 14.1 | 16.5 | 19.8 | 24.9 | 33.0 | 41.3 | 48.1 |
| 25kHz | 24.8 | 24.7 | 31.6 | 32.9 | 45.5 | 47.6 | 48.5 |
| 50kHz | 32.9 | 33.1 | 32.8 | 46.3 | 47.4 | 46.7 | 47.8 |
| 66kHz | 32.7 | 32.4 | 40.0 | 46.9 | 47.6 | 47.8 | 48.1 |

**Figure 12 – Reflector Tracking Updates per Second**

Further refinements of this device will be able to achieve significant performance increases. The searching speed could be vastly improved by using an intelligent optimized search routine, by using an automatic scanning mirror system, or by rewriting the existing code into a custom FPGA design, possibly supported by a high speed DSP processor. Additionally, modifying the communication board to accept Ethernet commands rather than the existing serial I/O would yield drastic speed improvements. Data rate is a highly limiting factor in the current setup, and improving this would increase speeds significantly. The control code could also be rewritten in a compiled language, such as a C based language, or even compiled directly using the Matlab Compiler, for more speed improvements.

# SECTION 2 – ANALYSIS OF CONTROL CODE

## Introduction to Code Analysis

All of the system control code has been written in the Matlab programming language, and it uses functionality from the Data Acquisition Toolbox companion product. The control program is comprised of approximately 800 lines of non-comment code, or about 1400 lines including comments. The code is heavily commented, both in overall function description, and inline commenting. This was a primary goal of the coding effort so that the code would be reusable and comprehensible by researchers to follow.

The code is comprised of a number of different types of functionality. These can be divided into a few basic categories, from low level to high level: hardware communication functions; initialization functions; command building functions; utility functions; high level functions; and the executive function. The low level code provides a basic set of functionality on which we can build the high level code. This greatly simplifies implementing a high level system structure into a easily manageable and maintainable code base. Complicated hardware tasks such as moving the laser to a specific X-Y coordinate or switching the laser color can be accomplished with a single high level function call which utilizes the middle and low level code blocks.

The next few sections will explain how the code is organized, what some of the main functions do, and how they are implemented. The ordering of the sections roughly correspond to the order they are executed during normal code flow.

## Code Analysis I - Initialization

The high level executive function is broken down into a few main sections. These include user defined variables, hardware initialization, hardware alignment, the search routine, and the track routine. The specifics on the code design of each of these sections will be described in later sections.

The main code flow begins with initialization of hardware settings and variables. Next the search routine is started. The search routine will search the entire field of regard for a modulating retro-reflector signal. Once found, it will start the tracking routine, which will attempt to keep the beam centered on the reflector indefinitely. If tracking is lost, it will reset the beam and begin searching for the modulator once again.

## Code Analysis II - Initialization

The initialization tasks are performed in a few different function calls. These include initGlobals, alignTool, setupSerialPort, and setupDAQPort. The initGlobals function initializes and defines the global variables. The alignTool function brings up a graphical alignment interface that allows the centering of the laser beams on the center of the gimbaled mirrors. The setupSerialPort function creates a Matlab serial port object, initializes its values to the required hardware settings, and readies the port for immediate communication. The setupDAQPort function creates a Matlab data acquisition object, initializes the acquisition rate and duration, and readies the port for immediate data retrieval. Each of these functions is described in detail below.

The initGlobals function sets up variables that will be static throughout the execution of the program. The `define` variable is the location all these variables will be kept. It can be thought of as a storage container for variables that need to be accessed from any block of code. It holds many subvariables which are used for purposes such as name aliases, scan point density, and commands sent to the TALP4020 board. Of the variables, a few merit special discussion.

The define.feedback variable changes the method the mirror scans to a new position. If set to true, it will utilize the internal optical feedback sensors on the mirror chip, and use a feedback loop to minimize time to move the mirror to its desired location and ensure the accuracy of its position. If set to `false`, it will bypass the mirror position sensor and directly control the DAC outputs on the mirror.

In the normal course of operation, it is set to utilize the feedback loop as the mirror response characteristics are superior. One minor flaw was discovered during testing, however. The position sensors on the mirror board are optical based sensors that utilize visible wavelengths for operation. This in itself is not a problem with the designed wavelength of the mirrors, but the visible light that this system uses can interfere with the position sensors, creating unstable oscillations. If the laser beam crosses into the gap between mirror and base, the position of the mirror becomes random, and beam lock is lost.

The feedback option completely disables the feedback loop, thus averting the aforementioned problem. However, controlling the mirror open loop is a slow, inaccurate, and problematic ordeal. The response time multiplies from 4ms (see diagram below) to approximately 400ms, depending on distance moved. The position of the mirror is inexact at best, as minor heat changes cause the current supplied to the mirror to fluctuate, thus making position dependent on temperature, an unwanted side effect.

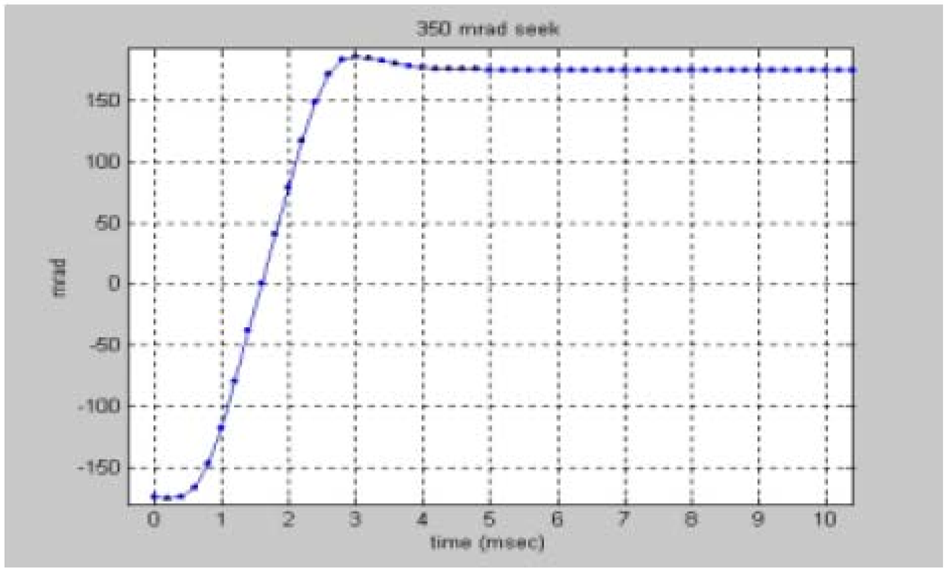


**Figure 13 - Response for full swing utilizing feedback loop [© Texas Instruments]**

The define.steps variable is used to select the point density of the scan. The scanner will scan this many points in each direction. In general, a scanning density of 20 to 40 was enough to detect the reflector at a two meter distance, using an abbreviated scanning region (approx ±4 degrees in each direction).

The alignTool function is a graphical interface tool used to move the first of the two MEMS mirrors into correct alignment with the second MEMS mirror. A simple interface allows moving in each of the four directions at different rates of movement. The goal of the tool is to align the beam with the center of the second mirror so that it can be reflected out into the world.

The setupSerialPort function attaches the serial port object to the computer's communications port, sets the data rate, and initializes the required hardware parameters. The serial port settings on the host computer need to match the parameters in the code exactly. This took some digging around the TALP source code to find, and the relevant parameters are listed here. The serial port uses 115,200 bits per second transmission rate, with 8 data bits and 1 stop bit. For error detection, it uses an "even" parity. This is a bit whose value is true if the number of "on" bits in the 8 bit data word is even. This way if a single bit's value is changed, the error can be detected. It does not utilize any flow control. Once these parameters are set up, data "packets" can be sent to the board via simple Matlab commands.

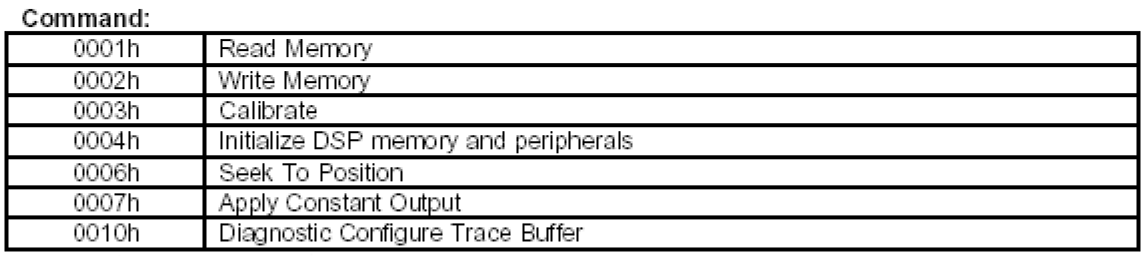
The setupDAQPort function initializes the National Instruments data acquisition card attached to the host PC. This card allows sensor readings to be passed from the acquirer to the code. In this way, the power and position detectors can be checked from software. The initialization function must be passed one input which tells it which channels on the DAQ board will be used. The SEEK mode allows for high frequency samples, with long durations. This allows for frequencies up to 100 kHz to be detected by Matlab accurately and reliably. The TRACK mode makes possible quick readings allowing for rapid responses to correct for any detected variation. The TRACK mode also collects multiple data channels, so that it can track both X and Y axis movement independently, whereas in the SEEK mode, we only care if we found anything, not where it is going.

These combinations of four pieces of code prepare the entire system for usage. The communications are set up, and the utility functions (e.g. seekTo) allow easy access to mirror movements. The sensor reading hardware is initialized, and can be accessed by the appropriate utility commands (e.g. getFFTPowData or getXYPowData). The mirrors have been aligned using the alignTool function, enabling the beam to be steered by the outer mirror. And finally, the global constants have been set so that other functions can operate with known parameters.

## Code Analysis III – Communication & Control Protocols

The TALP4020 board contains a high speed 16-bit DSP processor that works alongside an FPGA to provide communication, mirror steering, and diagnostic capability to the operator. Communication to and from the board uses a serial port with some nonstandard transmission settings.

Commands sent to the board over this serial port have a strict format. Each command structure consists of 32 words that are each 16 bits long. For readability, each 16 bit word is displayed in 4 hexadecimal digits. The first word in the command structure tells the board what type of command is being sent. The following words give the specifics relating to the command, such as where to move the mirrors, or what memory to read. Available command words are 0x0001 for a *memory read* command, 0x0002 for a *memory write* command, 0x0003 for a *calibrate mirror* command, 0x0004 for the *initialize memory and peripherals* command, 0x0006 for the *seek to position* command, 0x0007 for *direct mirror DAC control* command, and 0x0010 for a *diagnostic debugging* command. In the Matlab code, only the *seek to position* and the *direct DAC control* commands are used.



Command:

| | |
|---|---|
| 0001h | Read Memory |
| 0002h | Write Memory |
| 0003h | Calibrate |
| 0004h | Initialize DSP memory and peripherals |
| 0006h | Seek To Position |
| 0007h | Apply Constant Output |
| 0010h | Diagnostic Configure Trace Buffer |

**Figure 14 - Command Codes**

The 31 remaining words in each command structure vary widely among the above command types. The two commands extensively used in the code are discussed here. The 0x0006 (SEEK) command uses only 9 of the 31 available words. The first word (16 bits) is used to select the movement mode for each of the 4 mirrors that the board supports. Each mirror can be commanded to move to an absolute position for seeking, or to move to a position relative to the current position for tracking. Two bits for each mirror are used to choose between these movement modes, one for each axis. One bit is used to determine if an error message will be sent if an out of bounds position is sent to the mirror. The last bit is used to tell the processor if that mirror is active. Each mirror is then assigned an XY position using one word each for X and Y. Refer to the following table for a recap of these bits.

| Parm # | Description | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | Command number = 0006h | | | | | | | |
| 1 | Control flags | | | | | | | |
| | bit 15 | bit 14 | bit 13 | bit 12 | bit 11 | bit 10 | bit 9 | bit 8 |
| | APPLY3 | SAT3 | YAXIS3 | XAXIS3 | APPLY2 | SAT2 | YAXIS2 | XAXIS2 |
| | bit 7 | bit 6 | bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 |
| | APPLY1 | SAT1 | YAXIS1 | XAXIS1 | APPLY0 | SAT0 | YAXIS0 | XAXIS0 |
| 2 | Mirror 0 X-axis position. (mrad q9.7) | | | | | | | |
| 3 | Mirror 0 Y-axis position. (mrad q9.7) | | | | | | | |
| 4 | Mirror 1 X-axis position. (mrad q9.7) | | | | | | | |
| 5 | Mirror 1 Y-axis position. (mrad q9.7) | | | | | | | |
| 6 | Mirror 2 X-axis position. (mrad q9.7) | | | | | | | |
| 7 | Mirror 2 Y-axis position. (mrad q9.7) | | | | | | | |
| 8 | Mirror 3 X-axis position. (mrad q9.7) | | | | | | | |
| 9 | Mirror 3 Y-axis position. (mrad q9.7) | | | | | | | |

**Figure 15 - Seek to Position Command Structure**

The 0x0007 (DAC) command is very similar to the 0x0006 (SEEK) command. Each mirror again has 4 bits that describe the movement mode and 2 words for the XY position. The control bits consist of the same absolute/relative movement modes for each axis, and the same apply bit. The remaining bit differs from the previous command. The DAC has a fine resolution DAC for small movements, and a coarse resolution DAC for large movements. The control bit tells the board whether to adjust the coarse DAC for every command, or to keep the existing coarse value and just compensate with the fine value. Holding the coarse DAC will prevent large jumps in mirror position, but will take longer to reach the specified position. Refer to the table below for a recap of the different bits.

| Parm # | Description | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | Command number = 0007h | | | | | | | |
| 1 | Flags | | | | | | | |
| | bit 15 | bit 14 | bit 13 | bit 12 | bit 11 | bit 10 | bit 9 | bit 8 |
| | APPLY3 | MODE3 | YDAC3 | XDAC3 | APPLY2 | MODE2 | YDAC2 | XDAC2 |
| | bit 7 | bit 6 | bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 |
| | APPLY1 | MODE1 | YDAC1 | XDAC1 | APPLY0 | MODE0 | YDAC0 | XDAC0 |
| 2 | Mirror 0 X-axis DAC value. | | | | | | | |
| 3 | Mirror 0 Y-axis DAC value. | | | | | | | |
| 4 | Mirror 1 X-axis DAC value. | | | | | | | |
| 5 | Mirror 1 Y-axis DAC value. | | | | | | | |
| 6 | Mirror 2 X-axis DAC value. | | | | | | | |
| 7 | Mirror 2 Y-axis DAC value. | | | | | | | |
| 8 | Mirror 3 X-axis DAC value. | | | | | | | |
| 9 | Mirror 3 Y-axis DAC value. | | | | | | | |

**Figure 16 - Constant DAC Output Command Structure**

The commands are bundled together in the specified order by using the buildCommand function. Though only the previous two commands are actually used extensively in the code, all functionality of the board is contained in this function. This function inputs the command type requested, as well as up to 5 additional parameters required to build the specified command. Each command has an associated list of parameters with it. See Appendix X for details. The output of this function are the bytes, in order, that are sent to the board. Thus, the final stage of commanding the TALP board to point a mirror is to convert these commands into a format that is suitable to send across the serial port. To do this, we convert our vector of thirty-two16 bit binary commands into an array of 64 bytes. This array is 2 bytes wide (16 bit words) and 32 bytes long (32 words). Each byte pair is then sent across the serial cable in sequence to begin the command.

The byte stream is sent to the board by way of the sendCommand function. This function uses the standard C command fwrite, which usually writes data into a file, but in this case it writes it to the serial port. Optionally, the function will wait for the acknowledge command to be sent back to the host computer from the board. If an error code is returned, it will display the error message associated with the error code.

The buildCommand function combined with the sendCommand function is the basis for all communication within the program. Additionally, a second set of communication building functions was written to quickly modify the command structure. These functions, tweakSeekCommand and tweakDACCommand, only modify a small portion of an existing command structure. This is primarily useful in cases where the code is executing the same command numerous times sequentially, as is the case when performing a seek command. This is much faster (about 5x) than running the buildCommand script each time. These low level code blocks provide the functionality for easy running of the scanning and tracking code.

## Code Analysis IV – Search and Seek Algorithm

The search and seek algorithm's main function is to locate the reflector and then track it. It does this by utilizing the low level code blocks previously described to fully realize the search and seek algorithms. It will, on command, seek out a modulating or non-modulating reflector and proceed to track it using the position detector sensor. The function follows a simple line by line scan of the field of view, and records the reflected power at each point. The point or region of highest reflection (if any) is locked onto by the mirror. If more than one point returns high reflections, then a weighted center is found, and that point is locked onto by the mirror. Once a single point has been located as the most probable reflector location, control of the mirror pointing is passed to the tracking algorithm. The remainder of this section will describe in detail the seeking algorithm and its constituent functions.

The primary search routine is a simple raster scan of the covered field of view. In the code, the field of view is converted into a Cartesian grid, with the grid extents roughly corresponding to the ±11 degree field of view of the mirror. The **x** axis is defined as the yaw axis of the mirror (left or right), and the **y** axis is the pitch axis (up or down). For example, this leads to {**x**, **y**} = {0.5, -0.25} in the Cartesian grid mapping to [θ, φ] = [+5.5°, -2.75°] in the angular mirror coordinate system.

Since we are doing a simple XY raster scan, to scan the entire area we simply scan the Cartesian grid from -1 to 1 on each axis. However, due to mounting constraints and other limitations, the whole field of view is rarely entirely scanned. The positioning of the mirror on the table limits the placement of the reflector in roughly the top half of the coordinates. Also, since the mirror has quite large angular extents, the outer region often falls on regions far from where the reflector is located. However, this is beneficial to our system, since this means that the laser steering housing does not need very accurate alignment to point in the direction of the reflector.

Seeking from a long distance, i.e. more than a few meters, requires a fairly small scanning angle to cover a large physical area. The scanning resolution at which the mirror can move is of higher value to this system than the total area covered, since a higher resolution scanner can have a longer range. The TALP 1000A already has a very high resolution scanning capability, as is documented in the specifications. In the documentation, the commanded position sent to the board in Q9.7 format (See Appendix

X for Q Notation), and thus the minimum movement is $\frac{1}{2^7}$ mrad, or 7.8 μrad. This very high resolution would allow for 7.8 mm steps at a kilometer distance. This means that any retro reflecting modulator larger than 7.8 mm square will be detected from a kilometer away, assuming adequate power levels are used and divergence is minimized. This particular feature makes the system very attractive to current free space laser communication systems, as alignment and shifting are some of the main difficulties. This system would not only perform initial alignment, but would correct for any shifting that occurs.

To implement the scanning algorithm, two nested `for` loops are utilized. The outer loop increments the y axis, and the inner loop scans the x axis. This has the effect of starting at (-1, 1), the upper left corner, moving to (1, 1), the upper right corner, then moving back to the left, lowering the y value, and scanning to the right again. See figure X below for details.

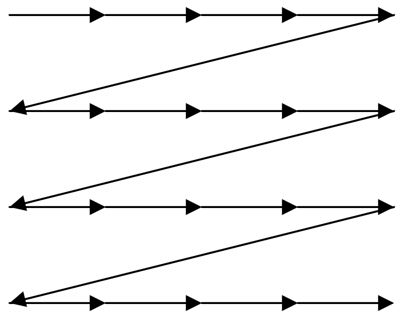


**Figure 17 - Raster Scan Pattern**

After each movement step, the reflected power is recorded into a two dimensional array, with each element in the array corresponding to a particular physical scan location. To capture the power level in this stage, we use the getFFTPoswData, depending on which of the two sensors is being used. This function collects a specified number of data points from the power detector, performs an FFT on the collected data, and returns the power level at the user-selected frequency. This function has been shown to effectively eliminate noise and ambient light power in lab tests.

After collecting the entire array of power reflection values, the point of highest return is calculated, and becomes the center point of the second scan. This new scan will have a much higher scanning resolution than the initial scan, and will limit itself to the region nearby to the maximum power point detected in the first scan.

The second scan is performed nearly identically to the first scan, except that the array of detected powers is processed slightly differently at the end of the scan. Because of the small steps, many points will have reflector hits, and thus a map of the reflective surface can be obtained. Using this map, we can find the centroid of the returned points, and approximate the center of the reflector. This point allows the tracking algorithm to correct for the highest magnitude displacement errors.

The centroid calculation routine follows a basic method of binary centroid detection. The center of any connected binary region is the point which has the least total distance to all other points in the region. To accomplish this, the two dimensional array of reflected powers is first normalized to 0 to 1. Then we mark any point whose power value is at least 60 percent of the maximum, e.g. 0.6 and above, as part of the centroid region. We then simply compute the total Euclidean distance of each point to every other point in the region. The point with the smallest total distance is the centroid. The x and y coordinates of this point are then returned. This point will be the center of the reflector, which will be used as the starting point of the tracking routine.

## Code Analysis V – Tracking Algorithm

The job of the tracking routine is to keep track of the reflector once it is found. This tracking routine makes heavy use of the beam position sensor to detect any lateral beam shifts, due to reflector movement. Any movement of the reflector will shift the beam in the opposite direction the same amount. A quick correction in beam position recenters the beam on the center of the reflector, thus tracking it.

The system utilizes an analog position sensing detector for aiding the tracker. The detector senses the position of the beam on the detection surface, and creates a signal whose voltage varies according to position. A separate channel is included for each axis. These voltages nominally span a 5V range centered around a point that is determined by the ambient light present. Moving the beam away from the center raises or lowers the voltage by a set amount per unit distance. By examining the changes in these voltages, we can calculate position changes in the incoming beam of light. This gives us a variable to track in the negative feedback control system.

The software of the tracking system utilizes a simple feedback loop. The voltage difference is converted to a position difference using a pre-calibrated constant. This position difference is sent to the mirror controller as a “relative move” command. This will cause the mirror to adjust the beam position to the new center of the reflector. The diagram below illustrates the feedback loop. The update time is currently fast enough to compensate for slow movements, approximately 0.25 to 1.0 cm/sec velocity. With some tweaking, this could be improved significantly. With dedicated hardware, this could be used to track extremely reliably and quickly.

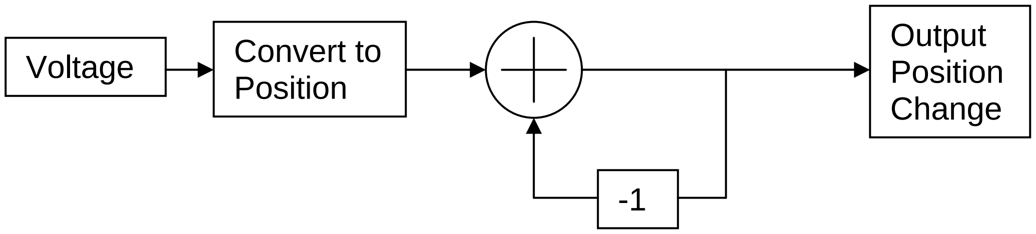


**Figure 18 - Tracking Feedback Loop**

In the current situation, it is possible to lose track of a rapidly moving device. To detect these losses of tracking, a signal power reading is taken each cycle and is compared to the original power level. If this drops below a given threshold, then the track is said to be lost, and the search routine is called again to find the device's new location.

## CONCLUSION

The system was successfully built and tested as originally envisioned. It was implemented by utilizing the power of the advanced modulating retroreflector and the steerable micro mirrors. The system was able to successfully find, track, and communicate with the remote receiver. This working prototype is ready to be developed further into either power distribution or wireless communication.

The original vision included using this as a stepping stone to future high performance designs. A number of improvements could be made to this implementation of the system to yield higher performance and smaller size. Modulation speed for the prototype system is limited to 50 kHz, though the modulating retroreflector is the main limiting factor here. This could be improved by utilizing a new modulator design, such as a thin arrayed diffractive modulator (whose development is already underway at UCSD). Seeking speed increase can be obtained by revamping the method used to control the mirrors, perhaps high speed optimized automated search routines loaded onto an FPGA or ASIC design, or by replacing the steerable micro-mirrors with a dedicated laser scanning system. Tracking ability can be increased by using a more robust set of sensors and cabling, as well as optimized tracking algorithm, perhaps one that includes future position prediction. Finally, size can be reduced by using smaller lasers, fewer reflecting optics, and perhaps a polarized beamsplitter.

Future designs can use this as a stepping stone to more advanced systems, possibly even revolutionizing the world by providing a solution to the last barrier in the quest to remove the remaining wires from consumer electronics.

# APPENDIX I – ABBREVIATIONS

| | |
|---|---|
| DAC | - Digital to Analog Converter |
| ADC | - Analog to Digital Converter |
| DAQ | - Data Acquisition Device |
| FFT | - Fast Fourier Transform |
| MEMS | - Micro Electro-Mechanical System |

# APPENDIX II – SOURCE CODE

## Source Code – Main Executive Function

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: talp4020SeekAndTrackV3                               %%%%%
%%%%% DESCRIPTION: This program will perform basic seeking and        %%%%%
%%%%%   tracking using these components:                             %%%%%
%%%%%   1. TALP4020 mirror control board                             %%%%%
%%%%%   2. 2 TALP1000A MEMS mirrors                                  %%%%%
%%%%%   3. National Instruments PCI 6024E 200kS/s DAQ board          %%%%%
%%%%%   4. serial cable connecting the PC to the TALP4020 board      %%%%%
%%%%%   5. Newport 1830-C with 818-SL detector head (power meter)    %%%%%
%%%%%   6. OnTrak OT301 with detector head (position sensing detector)%%%%%
%%%%%   7. National Instruments BNC-2090 (bnc to NI cable)           %%%%%
%%%%%   8. Custom optical system consisting of mirrors and lasers    %%%%%
%%%%%   9. Custom retro reflecting modulator                         %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
global define switchOnTrack

%-- User Inputs --%

steps = 30;                 % Scanning Density (steps by steps grid)
alignMode = 0;              % 0 = no alignment tool
                            % 1 = align orange laser
                            % 2 = align green laser
useTracking = 1;            % Enable tracking mode
useSeek = 1;                % Enable seeking mode
useLoop = 0;                % Enable continuous repeat of search/track
switchOnTrack = 1;          % switch color on tracking start
useCustomPosition = 0;      % Seek to a single position @ coords:
    xPos = 0;
    yPos = -.46;
freqToLookFor = 50e3;        % Frequency to scan for



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Scanning and Tracking Algorithm %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% Debugging statements Enable/Disable %%%
define.PRINTF = false;

fprintf('\n')

%%% Initialize global parameters %%%
load('mirrorSettings')
initGlobals(steps, freqToLookFor)

%%% Close any opened serial or DAQ ports %%%
close_serial_port
close_daq_port

%%% Open & initialize serial port %%%
s = setupSerialPort;
global waitForSerialReply
waitForSerialReply = true;

%%% Test serial port communications %%%
testSerialPort(s);
```

```
%%% Alignment Tool %%%
if alignMode == 1
    seekTo(s, 1, 0, 0)
    alignTool(s, define.ORANGE)
    return
elseif alignMode == 2
    seekTo(s, 1, 0, 0)
    alignTool(s, define.GREEN)
    return
end

%%% Set scanning mirror index %%%
mirror = 1;

%%% Ignore serial port replies to improve speed %%%
waitForSerialReply = false;

%%%%%%%%%%%%%%%%%%%%%%%
%%% Begin Main Loop %%%
%%%%%%%%%%%%%%%%%%%%%%%

% if switchOnTrack
%     for i = 1:10
%         %%% Switch a few times to check alignment
%         colorSelect(s, define.ORANGE);
%         pause(1)
%         colorSelect(s, define.GREEN);
%         pause(1)
%     end
% end

while 1

    %%% Select seeking laser %%%
    colorSelect(s, define.ORANGE);

    %%%%%%%%%%%%%%
    %%%  SEEK  %%%
    %%%%%%%%%%%%%%

    %%% Open & initialize seeking data acquisition port %%%
    d = setupDAQPort(define.Seek);

    if useCustomPosition
        seekTo(s, mirror, xPos, yPos)
    end

    if useSeek
        fprintf('Seeking...\n')
        tic
        rasterScanSeekV3(s, d, mirror);
        seekTime = toc
    end

    %%% Plot FFT Spectrum %%%
    [freq, power] = getFFTPowData(d,true);

    %%% Close seeking DAQ port %%%
    close_daq_port

    %%%%%%%%%%%%%
    %%% TRACK %%%
    %%%%%%%%%%%%%

%     if switchOnTrack
%         colorSelect(s, define.GREEN);
%     end

    %%% Open & initialize tracking data acquisition port %%%
    d = setupDAQPort(define.Track);

    if useTracking
        fprintf('Tracking...\n')
        tic
        trackPSDSensor(s, d, mirror);
        trackingTime=toc
    end

    %%% Close tracking DAQ port %%%
    close_daq_port

    pause(.5);

    if ~useLoop
        break
    end

end

% colorSelect(s, define.ORANGE);

%%% Close serial port %%%
close_serial_port
```

## Source Code – Main Seek Function

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: rasterScanSeek                                         %%%%%
%%%%% DESCRIPTION: This function will perform a raster scan along a   %%%%%
%%%%%   grid.  The grid density is contained in the define.steps      %%%%%
%%%%%   variable.  s is a pointer to the serial port, and d is a      %%%%%
%%%%%   pointer to the data acquisition port.                         %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function rasterScanSeek(s, d, mirrorPort)

global define
mirrorMode = define.NoSaturate;

%%% Build initial command %%%
cmd = buildCommand(define.seekToPos,mirrorPort,mirrorMode,define.ABS,...
    0, 0);

%%%%%%%%%%%%%%%%%%%
%%% GLOBAL SCAN %%%
%%%%%%%%%%%%%%%%%%%

    xmin = -.2;
    xmax = .9;
    ymin = -.6;
    ymax = -.1;

%%% Script to point the mirror at the corner points %%%
% testMaxima(s,cmd,mirrorPort,xmin,xmax,ymin,ymax)

%%% Calculate the step size between points %%%
xstep = abs(xmin-xmax)/define.steps;
ystep = abs(ymin-ymax)/define.steps;

%%% Create a vector of x and y points to scan %%%
xSteps = linspace(xmin,xmax,define.steps);
ySteps = linspace(ymax,ymin,define.steps);

%%% Perform large area scan %%%
[powerDetected] = scanArea(s,d,cmd,mirrorPort,xSteps,ySteps);

%%% Find the point of maximum power recorded %%%
[xLoc, yLoc] = findSensorLoc(powerDetected,false);
xPos = xSteps(xLoc);
yPos = ySteps(yLoc);

%%% Move to point of maximum power %%%
cmd = tweakSeekCommand(cmd,xPos,yPos,mirrorPort);
sendCommand(s,cmd);

%%%%%%%%%%%%%%%%%%
%%% LOCAL SCAN %%%
%%%%%%%%%%%%%%%%%%

%%% Calculate new points to scan %%%
fractionalStepSize = 1/4;
numStepsToScan = 2;
xstepSmall = xstep*fractionalStepSize;
ystepSmall = ystep*fractionalStepSize;
xSteps = (xPos-xstep*numStepsToScan):xstepSmall:(xPos+xstep*numStepsToScan);
```

```
ySteps = (yPos+ystep*numStepsToScan):-ystepSmall:(yPos-ystep*numStepsToScan);

%%% Perform small area scan %%%
[powerDetected] = scanArea(s,d,cmd,mirrorPort,xSteps,ySteps);

%%% Find the point of maximum power recorded %%%
[xLoc, yLoc] = findSensorLoc(powerDetected,true);
xPos = xSteps(xLoc);
yPos = ySteps(yLoc);

%%% Move to point of maximum power %%%
cmd = tweakSeekCommand(cmd,xPos,yPos,mirrorPort);
sendCommand(s,cmd);

fprintf('Sensor found at [%0.2f,%0.2f]!\n',xPos,yPos)

%-----------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: scanArea                                              %%%%%
%%%%% DESCRIPTION: This function scans the mirror along a raster scan %%%%%
%%%%%   grid using the x and y positions in xSteps and ySteps.        %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [powerDetected] = scanArea(s,d,cmd,mirrorPort,xSteps,ySteps)

%%% Get bias voltages if PSD is used for detection %%%
if length(d.channel) == 2
    [xBias, yBias] = getXYPowData(d);
end

%%% Begin scanning loop %%%
for yCount = 1:length(ySteps)
    for xCount = 1:length(xSteps)

        %%% Build seek command and send %%%
        cmd = tweakSeekCommand(cmd,xSteps(xCount), ySteps(yCount),mirrorPort);
        sendCommand(s,cmd);

        %%% Get power at selected point %%%
        if length(d.channel) == 1
            %%% Use this for power sensor %%%
            [f,p] = getFFTPowData(d);
            powerDetected(yCount,xCount) = p;
        else
            %%% Use this for position sensor %%%
            [x, y] = getXYPowData(d);
            [powerDetected(yCount,xCount,1) powerDetected(yCount,xCount,2)] = ...
                getVDiff(x,y,xBias,yBias);
        end

    end
end



%-----------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: testMaxima                                            %%%%%
%%%%% DESCRIPTION: This function will point the mirror at the four    %%%%%
%%%%%   corners of the input extrema.                                 %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function testMaxima(s,cmd,mirrorPort,xmin,xmax,ymin,ymax)
```

```
%%% Move to the corner points 5 times %%%
for i = 1:10
    %%% Top Left %%%
    cmd = tweakSeekCommand(cmd,xmin, ymax,mirrorPort);
    sendCommand(s,cmd);
    pause(1)

    %%% Top Right %%%
    cmd = tweakSeekCommand(cmd,xmax, ymax,mirrorPort);
    sendCommand(s,cmd);
    pause(1)

    %%% Bottom Right %%%
    cmd = tweakSeekCommand(cmd,xmax, ymin,mirrorPort);
    sendCommand(s,cmd);
    pause(1)

    %%% Bottom Left %%%
    cmd = tweakSeekCommand(cmd,xmin, ymin,mirrorPort);
    sendCommand(s,cmd);
    pause(1)
end

%--------------------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: findSensorLoc                                         %%%%%
%%%%% DESCRIPTION: This function inputs a maxtrix of voltage or power %%%%%
%%%%%   readings, and outputs the maximum reading location.  If       %%%%%
%%%%%   centroid is true, it will use the centroid discovery          %%%%%
%%%%%   algorithm below.                                              %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [xLoc, yLoc] = findSensorLoc(powerDetected, centroid)

if size(powerDetected,3) == 2
    %%% Extract x and y voltages %%%
    xPower = powerDetected(:,:,1);
    yPower = powerDetected(:,:,2);

    %%% Combine x and y voltages %%%
    sumPower = xPower + yPower;
else
    sumPower = powerDetected;
end

% %%% Plot image of power levels detected %%%
figure
imagesc(sumPower)
colormap(gray)
axis xy

%%% Get highest power point %%%
if centroid == false
    %%% Find point of maximum power %%%
    [yLoc, xLoc] = find(sumPower == max(max(sumPower)));
else
    %%% Find center of power that is 60% of maximum or bigger %%%
    fprintf('Scanning for centroid now...')
    [yLoc, xLoc] = findCentroid(sumPower);
    fprintf('done!\n')
end

%%% Eliminate multiple equal valued returns %%%
```

```
if length(xLoc)>1
    xLoc = xLoc(1);
    yLoc = yLoc(1);
end

%-----------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: findCentroid                                          %%%%%
%%%%% DESCRIPTION: This function inputs a maxtrix of numbers and does %%%%%
%%%%%   a few things:                                                 %%%%%
%%%%%   1. It normalizes the matrix to [0..1]                         %%%%%
%%%%%   2. Finds values above 60% of the maximum                      %%%%%
%%%%%   3. Finds the point with the smallest sum of distances to      %%%%%
%%%%%      other "on" pixels                                          %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [xLoc, yLoc] = findCentroid(in)

A = normalize(in);

B = (A>.60);

C = zeros(size(in));
for x = 1:size(in,2)
    for y = 1:size(in,1)
        C(y,x) = distToOnPixels(B,x,y);
    end
end

[xLoc, yLoc] = find( C == min(C(:)) );

%-----------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: distToOnPixels                                        %%%%%
%%%%% DESCRIPTION: This function inputs a maxtrix of booleans and     %%%%%
%%%%%   calculates the total distance to each "on" pixel              %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [totalDist] = distToOnPixels(in,x,y)

[onPixLocationsY, onPixLocationsX] = find(in);

totalDist = 0;
for i = 1:length(onPixLocationsX)
    totalDist = totalDist+ euclideanDistance(x,y,onPixLocationsX(i),onPixLocationsY(i));
end

%-----------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: euclideanDistance                                     %%%%%
%%%%% DESCRIPTION: This function finds the euclidean distance between %%%%%
%%%%%   two given points in xy space.                                 %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function dist = euclideanDistance(x1,y1,x2,y2)
dist = sqrt((x1-x2)^2 + (y1-y2)^2);
```

## Source Code – Main Track Function

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: trackPSDSensor                                        %%%%%
%%%%% DESCRIPTION: This function will track the position sensor       %%%%%
%%%%%   device using a simple negative feedback loop. It will quit    %%%%%
%%%%%   tracking if the signal moves off the sensor or drops too low  %%%%%
%%%%%   in power, or if the ambient light becomes too strong.         %%%%%
%%%%%   NOTE: Modulation is ignored for tracking                      %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function trackPSDSensor(s, d, mirrorPort)

global define switchOnTrack

mirrorMode = define.Saturate;

isLocked = true;
isTracking = false;

cmd = buildCommand(define.seekToPos,mirrorPort,mirrorMode,define.REL,...
    0, 0);

%%% Calculate laser offset discrepency
if switchOnTrack
    realignLaser(cmd,s,d);
end

while isLocked

    %%% Get DAQ data and convert to mirror coordinate system
    [xVolts, yVolts, p] = getXYPowData(d);
    [xs, ys, isTracking] = convertVoltageToXY(xVolts, yVolts, isTracking);
    [xm, ym] = convertSensorToMirrorCoords(xs, ys);

    %%% Verify that the laser is still on the sensor
    isLocked = isOnSensor(p,s);
    % fprintf('## P:%+0.4f\n',p)

    if isLocked == 1
     %%% Debugging statements for outputting coordinates
        % fprintf('## Xv %+0.4f ## Yv %+0.4f ## P:%+0.4f\n',xVolts,yVolts,p)
        % fprintf('## Xs %+0.4f ## Ys %+0.4f ## Locked:%0.0f\n',xs,ys,isLocked)
        % fprintf('## Xm %+0.4f ## Ym %+0.4f ## Locked:%0.0f\n',xm,ym,isLocked)
        % fprintf('## Xcmd %+0.4f ## Ycmd %+0.4f ## Locked:%0.0f\n',-xm,-ym,isLocked)

        %%% Send new relative position based on sensor data
        cmd = tweakSeekCommand(cmd,-xm,-ym,mirrorPort);
        sendCommand(s,cmd);
    end

end

%--------------------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: realignLaser                                          %%%%%
%%%%% DESCRIPTION: This function adjusts the second laser for any     %%%%%
%%%%%   recent drift that has occurred between the two lasers.        %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function realignLaser(cmd,s,d)
global define

[x1Volts, y1Volts, p] = getXYPowData(d);
[xs1, ys1, dummy] = convertVoltageToXY(x1Volts, y1Volts, false);

colorSelect(s, define.GREEN);
[x2Volts, y2Volts, p] = getXYPowData(d);
[xs2, ys2, dummy] = convertVoltageToXY(x2Volts, y2Volts, true);

xOffset = xs2-xs1;
yOffset = ys2-ys1;

define.autoAlign.seek.green.x = define.autoAlign.seek.green.x - xOffset/100;
define.autoAlign.seek.green.y = define.autoAlign.seek.green.y + yOffset/250;
colorSelect(s, define.GREEN);

%---------------------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: convertVoltageToXY                                    %%%%%
%%%%% DESCRIPTION: This function converts the voltages from the       %%%%%
%%%%%   position sensor to a normalized sensor coordinate system.     %%%%%
%%%%%   It uses the initial laser location as the origin of the       %%%%%
%%%%%   coordinate system, and will move the laser back here when     %%%%%
%%%%%   tracking movement.                                            %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [xs, ys, isTracking] = convertVoltageToXY(xVolts,yVolts,isTracking)
persistent xCenter yCenter

%%% Save center data for later reference if this is first data point
if isTracking == false
    xCenter = xVolts;
    yCenter = yVolts;
    isTracking = true;
end

%%% Nominal voltage swing from end to end
swingVolts = 1;

%%% Take difference, and normalize
xs = (xVolts - xCenter)/swingVolts;
ys = -(yVolts - yCenter)/swingVolts;

%---------------------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: convertSensorToMirrorCoords                           %%%%%
%%%%% DESCRIPTION: This function converts the normalized sensor       %%%%%
%%%%%   coordinates to a best guess mirror coordinate.  Accuracy is   %%%%%
%%%%%   not that important here, as long as we are within an order    %%%%%
%%%%%   of magnitude or so.                                           %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [xm, ym] = convertSensorToMirrorCoords(xs, ys)

customGain = 1/200;

%%% Positions are reversed in mirrors
xs = -xs;
ys = -ys;

%%% Mirror range is -192 to +192 mrad (+-11deg)= -1 munit to +1 munit
%%% eg 1 munit = 192 mrad

%%% Sensor is 20mm diam, so 1 sunit = 10mm
sunit_to_mm = 10;
mm_to_sunit = 1/sunit_to_mm;

%%% Fixed for mirror
z_munit = 1/tan(.192);        %%% Distance to wall in munits

%%% Path to mirror and back is ~4.4m
% z_mm = 4400/2;                %%% Distance in mm
z_mm = 30+ 37;
z_sunit = z_mm *mm_to_sunit;  %%% Distance in sunits

%%% Take ratio of two distances
munit_to_sunit = z_sunit/z_munit;
sunit_to_munit = 1/munit_to_sunit;

%%% Apply conversion factor
xm = xs*sunit_to_munit *customGain;
ym = ys*sunit_to_munit *customGain;

%-------------------------------------------------------------------------
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: isOnSensor                                            %%%%%
%%%%% DESCRIPTION: This function checks to see if the power received  %%%%%
%%%%%   is larger than a given threshold.                             %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = isOnSensor(p,s)
global define switchOnTrack

vThreshold =  0.005;
% vThreshold =  0.75;

persistent numMisses
if isempty(numMisses)
    numMisses = 0;
end

persistent lastGood
if isempty(lastGood)
    lastGood = true;
end



if p>vThreshold
    out = 1;
    numMisses = 0;
    if ~lastGood
        if switchOnTrack
            colorSelect(s, define.GREEN);
        end
    end
else
    if switchOnTrack
        colorSelect(s, define.ORANGE);
    end
    lastGood = false;
    if numMisses > 50
        out = 0;
    else
        out = 2;
        numMisses = numMisses+1;
    end
end
```

## Source Code – Initialization

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: initGlobals                                            %%%%%
%%%%% DESCRIPTION: This function will initialize the static global    %%%%%
%%%%%   variables used in the program.                                 %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function initGlobals(steps, freqToLookFor)

global define

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% Custom Definable  %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% Scanning Density %%%
define.steps = steps;

%%% Feedback Setting %%%
define.feedback = true;

%%% Frequency Scanning Setting %%%
define.frequency = freqToLookFor;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% System References %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% Color References %%%
define.GREEN = 0;
define.ORANGE = 1;

%%% Sensor Mode %%%
define.Seek = 0;
define.Track = 1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% Mirror References %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% DSP Commands %%%
define.readMemory               = 1;
define.writeMemory              = 2;
define.calibrate                = 3;
define.initMemoryAndPeripherals = 4;
define.seekToPos                = 6;
define.constantDACOutput        = 7;
define.configTraceBuffer        = 10;

%%% Read Memory Flags %%%
define.dataMemory    = 0;
define.programMemory = 1;
define.IOMemory      = 2;
define.useAddress    = 0;
define.blockRead     = 1;

%%% Calibration Mode Flags %%%
define.CTRL = 12:16;         %Recalculate control gains based on calibrated values
define.RES  = [11 13:16];    %Measure resonant frequency of spring/mass system
define.DAC  = [11:12 14:16]; %Cancel offsets in DAC and coil current criver
define.BIAS = [11:13 15:16]; %Measure spring bias force vs. displacement
define.DEF  = [11:14 16];    %Measure deflection range of mirror
define.IFB  = 11:15;       %Calibrate gain of internal feedback sensor
define.FULL = [];            %All calibration commands

%%% Movement Modes %%%   (DAC and SEEK)
define.ABS = 1;
define.REL = 0;

%%% Coarse DAC Mode %%%  (DAC)
define.recenter = 0;
define.holdCoarse = 1;

%%% Seek Mode %%%        (SEEK)
define.NoSaturate = 0;
define.Saturate = 1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: setupSerialPort                                       %%%%%
%%%%% DESCRIPTION: This function sets up the serial port to           %%%%%
%%%%%   communicate with the TALP4020 board. It uses a baud rate      %%%%%
%%%%%   of 115 kbps, with 8 data bits, 1 stop bit and even parity.    %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function s = setupSerialPort
global s

%%% Set up serial port
s = serial('COM1');
set(s,'BaudRate',115200);
set(s,'DataBits',8);

%%% Set up buffer to store incoming commands
s.InputBufferSize = 50000;

%%% Open serial port
fopen(s);
fprintf('Serial port opened!\n')

%%% Set remaining parameters
set(s,'Parity','even');
set(s,'StopBits',1);
set(s,'FlowControl','none');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: setupDAQPort                                          %%%%%
%%%%% DESCRIPTION: This function sets up the DAQ device to two        %%%%%
%%%%%   different modes.  The first mode will be a high speed         %%%%%
%%%%%   acquisition mode for the seeking routine, and the second      %%%%%
%%%%%   mode will be a low speed mode for tracking.                   %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function d = setupDAQPort(type)

global define



%%% Create NI DAQ Object %%%
d = analoginput('nidaq',1);

%%% Set input type to DAQ %%%
d.InputType = 'NonReferencedSingleEnded';
```

```
%%% Add channels to DAQ Object %%%
if type == define.Seek
    addchannel(d,2,'Power');    %% PWR
else
    addchannel(d,0,'XVal');  %% X
    addchannel(d,1,'YVal');  %% Y
    addchannel(d,2,'Power'); %% PWR
end

%%% Set up sampling rate %%%
% NI PCI 6024E is 200kS/s max rate %
if type == define.Seek
    d.SampleRate = 200e3;
    d.SamplesPerTrigger = 200;
else
    d.SampleRate = 66e3;
    d.SamplesPerTrigger = 50;
end

%%% Tell DAQ to sample immediately when called %%%
d.TriggerType = 'Immediate';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: testSerialPort                                        %%%%%
%%%%% DESCRIPTION: This function tests the serial port connection to  %%%%%
%%%%%   the TALP4020 board by attempting to read memory at 0x0020.    %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function testSerialPort(s)
global define

try
    addresses = hex2dec('0020');
    cmd =
buildCommand(define.readMemory,define.dataMemory,define.useAddress,1,addresses);
    sendCommand(s,cmd);
catch
    fprintf('WARNING SERIAL PORT COMMUNICATIONS FAILED!\n')
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% SCRIPT: close_serial_port                                       %%%%%
%%%%% DESCRIPTION: This script closes the serial port, deletes the    %%%%%
%%%%%   serial port object, and clears the object from memory.        %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if exist('s')
    if ~isempty(s)

        stopasync(s)
        fclose(s)
        delete(s)
        clear s

        disp('Serial port closed!')
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% SCRIPT: close_daq_port                                          %%%%%
%%%%% DESCRIPTION: This script stops the data acquisition, deletes    %%%%%
```

```
%%%%%   the data acqusition object, and clears the object from mem.   %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if exist('d')
    stop(d)
    delete(d)
    clear d

    fprintf('DAQ port closed!\n')
end
```

## Source Code – Command Functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: buildCommand                                          %%%%%
%%%%% DESCRIPTION: This function inputs a command mode and a number   %%%%%
%%%%%   of parameters associated with that command mode, and converts %%%%%
%%%%%   them into the appropriate commands that consist 32 16-bit     %%%%%
%%%%%   words.  Each command is then deconstructed into bytes to be   %%%%%
%%%%%   sent over the serial port. See                                %%%%%
%%%%%   "talp4020 software architecture 11-19-01.pdf" for details     %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function cmd = buildCommand(commandType,parm1,parm2,parm3,parm4,parm5)

global define

%%% Number of words in a command sequence %%%
cmdLength = 32;

%%% Create an empty command sequence %%%
cmd(1:cmdLength,1:16) = '0';

%%% Fill in the correct command type %%%
cmd(1,:) = dec2bin(commandType,16);

%%% Fill in the appropriate parameters %%%
switch commandType

    %%% Read from Memory Command %%%
    case define.readMemory
        cmd(2,13:14) = dec2bin(parm1,2);
        cmd(2,15:16) = dec2bin(parm2,2);
        cmd(3,:) = dec2bin(parm3,16);
        cmd(4,:) = dec2bin(parm4,16);
        if define.PRINTF
            fprintf('Read Memory command!\n')
        end



        %%% Write to Memory Command %%%
    case define.writeMemory
        %%% unused %%%



        %%% Calibration Command %%%
    case define.calibrate
        cmd(2,parm2) = '1';
        cmd(3,:) = dec2bin(parm1,16);
        fprintf('Calibrate command!\n')



        %%% Initialize Command %%%
    case define.initMemoryAndPeripherals
        %%% unused %%%



        %%% Seek to position using feedback command %%%
    case define.seekToPos
        flipXY = false;
        switch parm1 %mirrorPort
```

```
            case 0
                bits = 13:16;
                cmdRows = 3:4;
            case 1
                bits = 9:12;
                cmdRows = 5:6;
                flipXY = true;
            case 2
                bits = 5:8;
                cmdRows = 7:8;
            case 3
                bits = 1:4;
                cmdRows = 9:10;
        end

        if flipXY
            xSign = -1;
            ySign = -1;
        else
            xSign = 1;
            ySign = 1;
        end

        %%% Set submode parameters %%%
        cmd(2,bits(1)) = '1';               % APPLY
        cmd(2,bits(2)) = dec2bin(parm2,1);  % SAT          (sat/err)
        cmd(2,bits(3)) = dec2bin(parm3,1);  % Y-AXIS MODE (rel/abs)
        cmd(2,bits(4)) = dec2bin(parm3,1);  % X-AXIS MODE (rel/abs)

        %%% Convert the x and y position to Q notation %%%
        cmd(cmdRows(1),:) = float2qBin(9,7,convertX(parm4*xSign)); % X-AXIS
        cmd(cmdRows(2),:) = float2qBin(9,7,convertY(parm5*ySign)); % Y-AXIS



        %%% Seek to position by controlling DACs directly %%%
    case define.constantDACOutput
        flipXY = false;

        switch parm1 %mirrorPort
            case 0
                bits = 13:16;
                cmdRows = 3:4;
            case 1
                bits = 9:12;
                cmdRows = 5:6;
            case 2
                bits = 5:8;
                cmdRows = 7:8;
            case 3
                bits = 1:4;
                cmdRows = 9:10;
        end

        if flipXY
            xSign = -1;
            ySign = -1;
        else
            xSign = 1;
            ySign = 1;
        end

        %%% Set submode parameters %%%
        cmd(2,bits(1)) = '1';               % APPLY
        cmd(2,bits(2)) = dec2bin(parm2,1);  % MODE      (coarse)
        cmd(2,bits(3)) = dec2bin(parm3,1);  % YDAC mode (rel/abs)
        cmd(2,bits(4)) = dec2bin(parm3,1);  % XDAC mode (rel/abs)

        %%% Convert the x and y position to Q notation %%%
        cmd(cmdRows(1),:) = float2qBin(2,14,parm4*xSign); % XDAC
        cmd(cmdRows(2),:) = float2qBin(2,14,parm5*ySign); % YDAC



        %%% Configure Trace Buffer Command %%%
    case define.configTraceBuffer
        %%% unused %%%
end

%%% convert binary to hexidecimal to bytes %%%
cmd = bin2hex(cmd);
cmd = hex2bytes(cmd);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: tweakSeekCommand                                          %%%%%
%%%%% DESCRIPTION: This function inputs a pre-built seekToPos command %%%%%
%%%%%   structure and outputs a modified position command.             %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function cmd = tweakSeekCommand(cmd,xPos,yPos,mirror)

%%% Swap x and y axes for mirror index 1 %%%
if mirror == 1
    xPos = -xPos;
    yPos = -yPos;
end

%%% Select appropriate command rows to set mirror command %%%
if mirror == 0
    cmdrows = [3 4];
elseif mirror == 1
    cmdrows = [5 6];
end

%%% Convert to mirror coordinate system %%%
xPos = convertX(xPos);
yPos = convertY(yPos);

%%% 1. Change mirror coords to Q notation %%%
%%% 2. Change Q notated coords to bytes %%%
%%% 3. Update command structure %%%
cmd(cmdrows(1),:) = dec2bytes(float2qDec(9,7,xPos));
cmd(cmdrows(2),:) = dec2bytes(float2qDec(9,7,yPos));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: tweakDACCommand                                           %%%%%
%%%%% DESCRIPTION: This function inputs a pre-built DACToPos command  %%%%%
%%%%%   structure and outputs a modified position command.             %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function cmd = tweakDACCommand(cmd,xPos,yPos,mirror)

%%% Swap x and y axes for mirror index 1 %%%
if mirror == 1
    xPos = -xPos;
    yPos = -yPos;
end

%%% Select appropriate command rows to set mirror command %%%
```

```
if mirror == 0
    cmdrows = [3 4];
elseif mirror == 1
    cmdrows = [5 6];
end

%%% 1. Change mirror coords to Q notation %%%
%%% 2. Change Q notated coords to bytes %%%
%%% 3. Update command structure %%%
cmd(cmdrows(1),:) = dec2bytes(float2qDec(2,14,xPos));
cmd(cmdrows(2),:) = dec2bytes(float2qDec(2,14,yPos));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: sendCommand                                           %%%%%
%%%%% DESCRIPTION: This function writes the inputted command to the   %%%%%
%%%%%   serial port.  If there is an error, it tries up to 100 times  %%%%%
%%%%%   before quitting.  If it is commanded to wait for the          %%%%%
%%%%%   acknowledgement, it will do so, and parse the reply           %%%%%
%%%%%   accordingly.                                                  %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function sendCommand(s,cmd)
global waitForSerialReply

if size(cmd,1) > 1 && size(cmd,2) > 1
    cmd = cmd';
    cmd = cmd(:);
end

%%% Send Command %%%
for i = 1:50
    try
        fwrite_TALP(s,cmd)
        break
    catch
        %%% Display error if generated %%%
        disp(lasterr)
    end
end

%%% Wait for acknowledgement if requested %%%
if waitForSerialReply
    while s.BytesAvailable == 0
        pause(.010)
    end

    rc = fread(s,s.BytesAvailable,'uint8');

    for i = 1:length(rc)/2
        idx = (i*2-1):(i*2);
        disp(parseRC(rc(idx)'));
    end
end
```

## Source Code – Mirror Movement Functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: seekTo                                                %%%%%
%%%%% DESCRIPTION: This function is a quick command to move the       %%%%%
%%%%%   mirror to the specified position.It uses closed loop control. %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function seekTo(s, mirrorPort, xPos, yPos)

global define
persistent cmd

mirrorMode = define.Saturate;
if isempty(cmd)
    %%% No commands sent yet
    cmd = buildCommand(define.seekToPos,mirrorPort,mirrorMode,define.ABS,...
        xPos, yPos);
elseif cmd(1,2) == 6
    %%% Previous command was seekToPos
    if mirrorPort == 1
        cmd = tweakSeekCommand(cmd,xPos, yPos, true);
    else
        cmd = tweakSeekCommand(cmd,xPos, yPos, false);
    end
else
   %%% Some other command sent
    cmd = buildCommand(define.seekToPos,mirrorPort,mirrorMode,define.ABS,...
        xPos, yPos);
end
sendCommand(s,cmd);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: DACTo                                                 %%%%%
%%%%% DESCRIPTION: This function is a quick command to move the       %%%%%
%%%%%   mirror to the specified position.  It uses open loop control. %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function DACTo(s, mirrorPort, xPos, yPos)

global define
persistent cmd

mirrorMode = define.NoSaturate;
if isempty(cmd)
    %%% No commands sent yet
    cmd = buildCommand(define.constantDACOutput,mirrorPort,mirrorMode,define.ABS,...
        xPos, yPos);
elseif cmd(1,2) == 6
    %%% Previous command was seekToPos
    if mirrorPort == 1
        cmd = tweakDACCommand(cmd,xPos, yPos, true);
    else
        cmd = tweakDACCommand(cmd,xPos, yPos, false);
    end
else
   %%% Some other command sent
    cmd = buildCommand(define.constantDACOutput,mirrorPort,mirrorMode,define.ABS,...
        xPos, yPos);
end
sendCommand(s,cmd);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: colorSelect                                          %%%%%
%%%%% DESCRIPTION: This function moves the first mirror to the pre-   %%%%%
%%%%%   computed position for the inputted color.                    %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function colorSelect(s, color)
global define

%%% Get x and y positions %%%
if color == define.ORANGE
    if define.feedback
        xPos = define.autoAlign.seek.orange.x;
        yPos = define.autoAlign.seek.orange.y;
    else
        xPos = define.autoAlign.dac.orange.x;
        yPos = define.autoAlign.dac.orange.y;
    end
else
    if define.feedback
        xPos = define.autoAlign.seek.green.x;
        yPos = define.autoAlign.seek.green.y;
    else
        xPos = define.autoAlign.dac.green.x;
        yPos = define.autoAlign.dac.green.y;
    end
end

%%% Select color changing mirror %%%
mirror = 0;

%%% Move mirror to selected position %%%
if define.feedback
    cmd = buildCommand(define.seekToPos,mirror,define.Saturate,define.ABS,...
        xPos, yPos);
else
    cmd = buildCommand(define.constantDACOutput,mirror,define.Saturate,define.ABS,...
        xPos, yPos);
end
sendCommand(s,cmd);

pause(.1)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: returnToCenter                                        %%%%%
%%%%% DESCRIPTION: This function commands the speficied mirror to     %%%%%
%%%%%   {0, 0}                                                       %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function returnToCenter(s, mirrorPort)
global define

if define.feedback
    seekTo(s,mirrorPort, 0, 0);
else
    DACTo(s,mirrorPort, 0, 0);
end
```

## Source Code – Power Detection Functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: getFFTPowData                                        %%%%%
%%%%% DESCRIPTION: This function requests the current data from the   %%%%%
%%%%%   DAQ board, calculates the FFT of it, and outputs the power    %%%%%
%%%%%   at the requested frequency.                                   %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [freq, power] = getFFTPowData(d, doPlot)
global define

%%% Initialize Stuff %%%
if ~exist('doPlot')
    doPlot = false;
end
usedB = false;
requestedFrequency = define.frequency;

%%% Collect Data %%%
start(d)
data = getdata(d);

%%% Setup FFT Spacing %%%
Fs = d.SampleRate;
N = length(data);
df = Fs/N;
freqs = 0:df:(Fs-df);

%%% Perform FFT on Data %%%
data_mag_fft = abs(fft(data));

if requestedFrequency ~= 0
    %%% If looking for nonzero frequency %%%

    %%% Only search non DC frequencies %%%
    if usedB
        data_noDC_noAliasing = 10*log10(data_mag_fft(2:N/2-1)/N);
    else
        data_noDC_noAliasing = data_mag_fft(2:N/2-1)/N;
    end
    freqs_noDC_noAliasing = freqs(2:N/2-1);

    %%% Find frequency closest to requested %%%
    [tmpFreqDiffVector, indices] = sort(abs(freqs_noDC_noAliasing-requestedFrequency));
    %idx = find(tmpFreqDiffVector == min(abs(tmpFreqDiffVector)));
    idx = indices(1);

    %%% Report frequency closest to request and power at that freq %%%
    freq = freqs_noDC_noAliasing(idx);
    power = data_noDC_noAliasing(idx);

    %%% If plot requested, do plot %%%
    if doPlot
        % %%% Plot %%%
        figure;
        plot(freqs_noDC_noAliasing/1e3, data_noDC_noAliasing);
        xlabel('Frequency [kHz]')
        ylabel('Power')
        % axis([freqs_noDC_noAliasing(1)/1e3  freqs_noDC_noAliasing(end)/1e3, ...
        %          0 .1])
        fprintf('%0.0f Hz power at receiver is %2.2f\n',freq,power);

    end

else
    %%% Look for DC only %%%

    freq = 0;
    power = data_mag_fft(1);

    if doPlot
        fprintf('DC power at receiver is %2.2f\n',power);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: getXYPowData                                          %%%%%
%%%%% DESCRIPTION: This function runs the DAQ for the preset time and %%%%%
%%%%%   collects the voltages from the PSD and power meter.           %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [x, y, p] = getXYPowData(d)

%%% Run DAQ
start(d)

%%% Collect data from DAQ buffer
data = getdata(d);

%%% Take average value
avgdata = mean(data);

%%% Partition out the channels into respective variables
x = avgdata(1);
y = avgdata(2);
p = avgdata(3);
```

## Source Code – Binary, Hexadecimal, Byte, and Q Conversion Functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: bin2hex                                               %%%%%
%%%%% DESCRIPTION: This function converts a 16bit binary number to    %%%%%
%%%%%   hexadecimal format.                                           %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = bin2hex(in)

out = dec2hex(bin2dec(in),4);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: bytes2bin                                             %%%%%
%%%%% DESCRIPTION: This function converts decimal formatted bytes     %%%%%
%%%%%   to 16bit binary format.                                       %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = bytes2bin(in)

numWords = size(in,1);
numBytes = 2;

out(1:numWords,1:16) = '0';

odds = 1:2:numBytes;
evens = 2:2:numBytes;
out(:,1:8) = dec2bin(in(:,odds),8);
out(:,9:16) = dec2bin(in(:,evens),8);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: dec2bytes                                             %%%%%
%%%%% DESCRIPTION: This function inputs an integer (max 16 bits),     %%%%%
%%%%%   split it up into 8bit sections, and convert those back to     %%%%%
%%%%%   integer form.                                                 %%%%%
%%%%%   Example:  0000111100000101 -> 00001111, 00000101 -> 15, 5     %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = dec2bytes(in)

upperhalf = floor(in/2^8);
lowerhalf = rem(in,2^8);

out = [upperhalf lowerhalf];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: hex2bytes                                             %%%%%
%%%%% DESCRIPTION: This function converts a 4 hex string into its     %%%%%
%%%%%   2 constituent bytes, in decimal format.                       %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = hex2bytes(in)

numWords = size(in,1);
numBytes = size(in,2)/2;

out = zeros(numWords,numBytes);

odds = 1:2:numBytes;
```

```
evens = 2:2:numBytes;
out(:,odds) = hex2dec(in(:,1:2));
out(:,evens) = hex2dec(in(:,3:4));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: float2qBin                                              %%%%%
%%%%% DESCRIPTION: This function inputs a number and converts it to      %%%%%
%%%%%   the specified a.b Q notation. Output is binary.                 %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = float2qBin(a,b,num)

if num>=0
    out = dec2bin(num*2^b,16);
else
    if round(-num*2^b) == 0
        out = dec2bin(0,16);
    else
        out = dec2bin(bitcmp(round(-num*2^b),15)+1,15);
        out = ['1' out];
    end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: float2qDec                                              %%%%%
%%%%% DESCRIPTION: This function inputs a number and converts it to      %%%%%
%%%%%   the specified a.b Q notation. Output is decimal.                %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = float2qDec(a,b,num)

if num>=0
    out = fix(num*2^b);
else
    out = fix((bitcmp(round(-num*2^b),15)+1)+2^15);
end
```

## Source Code – Mirror Coordinate Transformation Functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: convertX                                              %%%%%
%%%%% DESCRIPTION: Converts normalized planar coordinates (e.g.       %%%%%
%%%%%   cartesian coordinates) into the mirror coordinate system      %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function out = convertX(in)

if abs(in)>1
    if in>1
        in = 1;
    else
        in = -1;
    end
end
out = -(in*192);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: convertY                                              %%%%%
%%%%% DESCRIPTION: Converts normalized planar coordinates (e.g.       %%%%%
%%%%%   cartesian coordinates) into the mirror coordinate system      %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function out = convertY(in)

if abs(in)>1
    if in>1
        in = 1;
    else
        in = -1;
    end
end
out = -(in*192);
```

## Source Code – Utility Functions

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: parseRC                                                 %%%%%
%%%%% DESCRIPTION: This function inputs a TALP4020 error code and       %%%%%
%%%%%   outputs the message corresponding with that code.               %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function msg = parseRC(rc)

%%% Convert code to hex %%%
rcBin = bytes2bin(rc);
rcHex = bin2hex(rcBin);

%%% Check to see which code is returned %%%
switch rcHex
    case '0000'
        %msg = '    CMD_SUCCESS';
        msg = [];
    case '0031'
        msg = '    ERR_CAL_INVALID_MIRROR';
    case '0032'
        msg = '    ERR_IPF_CAL_FAILED_LOW';
    case '0033'
        msg = '    ERR_IPF_CAL_FAILED_HIGH';
    case '0034'
        msg = '    ERR_CAL_BIAS_RANGE';
    case '0340'
        msg = '    ERR_CAL_BIAS_RANGE: M0 XAXIS';
    case '0341'
        msg = '    ERR_CAL_BIAS_RANGE: M0 YAXIS';
    case '0342'
        msg = '    ERR_CAL_BIAS_RANGE: M1 XAXIS';
    case '0343'
        msg = '    ERR_CAL_BIAS_RANGE: M1 YAXIS';
    case '0344'
        msg = '    ERR_CAL_BIAS_RANGE: M2 XAXIS';
    case '0345'
        msg = '    ERR_CAL_BIAS_RANGE: M2 YAXIS';
    case '0346'
        msg = '    ERR_CAL_BIAS_RANGE: M3 XAXIS';
    case '0347'
        msg = '    ERR_CAL_BIAS_RANGE: M3 YAXIS';
    case '003F'
        msg = '    ERR_CAL_CAL_NOT_SUPPORTED';
    case '0061'
        msg = '    ERR_SEEK_INVALID_PARM';
    case '0062'
        msg = '    ERR_SEEK_RANGE_EXCEEDED';
    case '0063'
        msg = '    ERR_SEEK_TIMEOUT';
    case '0071'
        msg = '    ERR_SEEK_INVALID_PARM';
    otherwise
        msg = ['    Unknown error code: 0x', rcHex];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: normalize                                               %%%%%
%%%%% DESCRIPTION: This function maps a vector to 0 to 1.               %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function out = normalize(in)

out = in / max(in(:));
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: getVDiff                                               %%%%%
%%%%% DESCRIPTION: This function simply finds the distance between    %%%%%
%%%%%   x and xBias, as well as y and yBias.                           %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xDiff, yDiff] = getVDiff(x,y,xBias,yBias)

xDiff = abs(x - xBias);
if ~isempty(y) && ~isempty(yBias)
    yDiff = abs(y - yBias);
else
    yDiff = [];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: fwrite_TALP                                            %%%%%
%%%%% DESCRIPTION: This function is a modified version of the         %%%%%
%%%%%   standard fwrite function.  It eliminates costly error checks  %%%%%
%%%%%   and switch statements since the input is known for the TALP.  %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function fwrite_TALP(s, cmd)

%    MP 7-13-99
%    Copyright 1999-2002 The MathWorks, Inc.
%    $Revision: 1.6 $  $Date: 2002/03/15 00:14:42 $

%    Modified for the specific TALP 4020 Board

mode = 0;

cmd = uint8(cmd);
type = 5;
signed = 0;

% Call the write java method.
try
   fwrite(igetfield(s, 'jobject'), cmd, length(cmd), type, mode, signed);
catch
   rethrow(lasterror);
end
```

## Source Code – Manual Alignment Tool

**NOTE: This code runs with a binary GUI file, alignTool.fig, and will function only when that file is present**

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%% FUNCTION: alignTool                                            %%%%%
%%%%% DESCRIPTION: This function will create a UI for moving the      %%%%%
%%%%%   color changing mirror.                                       %%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function varargout = alignTool(varargin)
% ALIGNTOOL M-file for alignTool.fig
%      ALIGNTOOL, by itself, creates a new ALIGNTOOL or raises the existing
%      singleton*.
%
%      H = ALIGNTOOL returns the handle to a new ALIGNTOOL or the handle to
%      the existing singleton*.
%
%      ALIGNTOOL('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in ALIGNTOOL.M with the given input arguments.
%
%      ALIGNTOOL('Property','Value',...) creates a new ALIGNTOOL or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before alignTool_OpeningFunction gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to alignTool_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help alignTool

% Last Modified by GUIDE v2.5 25-Oct-2004 14:35:43

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @alignTool_OpeningFcn, ...
                   'gui_OutputFcn',  @alignTool_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT



% --- Executes just before alignTool is made visible.
function alignTool_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to alignTool (see VARARGIN)

% Choose default command line output for alignTool
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes alignTool wait for user response (see UIRESUME)
% uiwait(handles.figure1);

global data define

data.s = varargin{1};
data.color = varargin{2};
if isfield(define, 'autoAlign')
    if data.color == define.ORANGE
        try
            if define.feedback
                data.x = define.autoAlign.seek.orange.x;
                data.y = define.autoAlign.seek.orange.y;
            else
                data.x = define.autoAlign.dac.orange.x;
                data.y = define.autoAlign.dac.orange.y;
            end
        catch
            data.x = 0;
            data.y = 0;
        end
    else
        try
            if define.feedback
                data.x = define.autoAlign.seek.green.x;
                data.y = define.autoAlign.seek.green.y;
            else
                data.x = define.autoAlign.dac.green.x;
                data.y = define.autoAlign.dac.green.y;
            end
        catch
            data.x = 0;
            data.y = 0;
        end
    end
else
    data.x = 0;
    data.y = 0;
end
data.mirror = 0;
moveMirror;




% --- Outputs from this function are returned to the command line.
function varargout = alignTool_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;
```

```
% --- Executes on button press in L3.
function L3_Callback(hObject, eventdata, handles)
% hObject    handle to L3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.x = data.x - 0.1;
moveMirror;
updateFigure(handles);



% --- Executes on button press in L2.
function L2_Callback(hObject, eventdata, handles)
% hObject    handle to L2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.x = data.x - 0.01;
moveMirror;
updateFigure(handles);



% --- Executes on button press in L1.
function L1_Callback(hObject, eventdata, handles)
% hObject    handle to L1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.x = data.x - 0.001;
moveMirror;
updateFigure(handles);



% --- Executes on button press in U1.
function U1_Callback(hObject, eventdata, handles)
% hObject    handle to U1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.y = data.y + 0.001;
moveMirror;
updateFigure(handles);



% --- Executes on button press in U2.
function U2_Callback(hObject, eventdata, handles)
% hObject    handle to U2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.y = data.y + 0.01;
moveMirror;
updateFigure(handles);



% --- Executes on button press in U3.
function U3_Callback(hObject, eventdata, handles)
% hObject    handle to U3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.y = data.y + 0.1;
moveMirror;
```

```
updateFigure(handles);



% --- Executes on button press in D1.
function D1_Callback(hObject, eventdata, handles)
% hObject    handle to D1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.y = data.y - 0.001;
moveMirror;
updateFigure(handles);



% --- Executes on button press in D2.
function D2_Callback(hObject, eventdata, handles)
% hObject    handle to D2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.y = data.y - 0.01;
moveMirror;
updateFigure(handles);



% --- Executes on button press in D3.
function D3_Callback(hObject, eventdata, handles)
% hObject    handle to D3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.y = data.y - 0.1;
moveMirror;
updateFigure(handles);



% --- Executes on button press in R1.
function R1_Callback(hObject, eventdata, handles)
% hObject    handle to R1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.x = data.x + 0.001;
moveMirror;
updateFigure(handles);



% --- Executes on button press in R2.
function R2_Callback(hObject, eventdata, handles)
% hObject    handle to R2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
global data
data.x = data.x + 0.01;
moveMirror;
updateFigure(handles);



% --- Executes on button press in R3.
function R3_Callback(hObject, eventdata, handles)
% hObject    handle to R3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
global data
data.x = data.x + 0.1;
moveMirror;
updateFigure(handles);



% --- Executes on button press in done.
function done_Callback(hObject, eventdata, handles)
% hObject    handle to done (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

global data define

if data.color == define.ORANGE
    if define.feedback
        define.autoAlign.seek.orange.x = data.x;
        define.autoAlign.seek.orange.y = data.y;
    else
        define.autoAlign.dac.orange.x = data.x;
        define.autoAlign.dac.orange.y = data.y;
    end
else
    if define.feedback
        define.autoAlign.seek.green.x = data.x;
        define.autoAlign.seek.green.y = data.y;
    else
        define.autoAlign.dac.green.x = data.x;
        define.autoAlign.dac.green.y = data.y;
    end
end
save('mirrorSettings','define')




% Get the current position of the GUI from the handles structure
% to pass to the modal dialog.
pos_size = get(handles.figure1,'Position');
% Call modaldlg with the argument 'Position'.
% user_response = modaldlg('Title','Confirm Close');
% switch user_response
% case {'No'}
%     % take no action
% case 'Yes'
%     % Prepare to close GUI application window
%     %                  .
%     %                  .
%     %                  .
    delete(handles.figure1)
% end




function moveMirror
global define data

if define.feedback
    cmd = buildCommand(define.seekToPos,data.mirror,define.Saturate,define.ABS,...
        data.x, data.y);
else
    cmd =
buildCommand(define.constantDACOutput,data.mirror,define.Saturate,define.ABS,...
        data.x, data.y);
end

sendCommand(data.s,cmd);

function updateFigure(handles)
global data
set(handles.xval,'String',num2str(data.x,3))
set(handles.yval,'String',num2str(data.y,3))



% --- Executes during object creation, after setting all properties.
function xval_CreateFcn(hObject, eventdata, handles)
% hObject    handle to xval (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end




function xval_Callback(hObject, eventdata, handles)
% hObject    handle to xval (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of xval as text
%        str2double(get(hObject,'String')) returns contents of xval as a double



% --- Executes during object creation, after setting all properties.
function yval_CreateFcn(hObject, eventdata, handles)
% hObject    handle to yval (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end




function yval_Callback(hObject, eventdata, handles)
% hObject    handle to yval (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of yval as text
%        str2double(get(hObject,'String')) returns contents of yval as a double
% --- Executes on button press in checkOrange.
function checkOrange_Callback(hObject, eventdata, handles)
% hObject    handle to checkOrange (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of checkOrange

set(handles.checkGreen,'Value',~get(hObject,'Value'))

% --- Executes on button press in checkGreen.
function checkGreen_Callback(hObject, eventdata, handles)
% hObject    handle to checkGreen (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hint: get(hObject,'Value') returns toggle state of checkGreen

set(handles.checkOrange,'Value',~get(hObject,'Value'))
```

# BIBLIOGRAPHY

Texas Instruments, TALP4020EVM Software Architecture Reference Guide, [PDF] November 19, 2001

Texas Instruments, TALP1000A Optical Networking Analog Mirror, [PDF] March 13, 2002

The MathWorks, Matlab User Documentation, [Online] Available http://www.mathworks.com/access/helpdesk/help/helpdesk.html, October 31, 2008

Chan, Trevor.  MEMS Mirror Array for Retroreflecting Modulators, [PPT Presentation] May 27, 2004