New Laser Rangefinder Designs for Tracking Hands Atop Large Interactive Surfaces

By

Christopher Yang

B.S., Electrical Engineering and Computer Science (1999)

Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Computer Science and Electrical Engineering

At the

Massachusetts Institute of Technology

March 2001

© 2001 Massachusetts Institute of Technology. All rights reserved

Signature of Author . . . . . .
Department of Electrical Engineering and Computer Science
March 1, 2001

Certified by . . . . . .
Joseph A. Paradiso
Principal Research Scientist, Media Arts and Sciences
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

New Laser Rangefinder Designs for Tracking Hands Atop Large Interactive Surfaces

By

Christopher Yang

Submitted to the Department of Electrical Engineering and Computer Science
On March 1, 2001 in partial fulfillment of the requirements for the
Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

As computers become increasingly integrated into our daily lives, ways of interacting with them must also become more natural. For example, large video walls are already common in public spaces, but while many are driven by digital video streams, all are passive and accommodate no natural mode of interaction. In order to make them responsive (like video kiosks) bare hand motions must be tracked atop large interactive surfaces. To this end, an inexpensive portable laser rangefinder prototype was developed by our group in 1998. This thesis tracks the development of a smaller, lighter, and more accessible scanning head better suited to practical application. In addition, two sets of baseband electronics were built. The first was based on previous designs, and altered to interface with the new scanning head. The second device was based around a much more compact single motherboard and used a new single-chip microcomputer with increased A/D resolution. With this system, hands are tracked by ranging from the planar-scanned laser. As a result, this technique suffers from occlusion with multiple hands (e.g. when two or more hands are aligned with the laser, the first hand is shadowed). This thesis finishes with a potential solution to this problem, showing a two-handed application to demonstrate a multi-handed workaround approach.

Thesis Supervisor: Joseph A. Paradiso
Title: Principal Research Scientist, MIT Media Laboratory

Acknowledgements

To **Joe Paradiso**, who took a giant leap of faith in taking me on for this project, and taught me everything from soldering to engineering to writing. Always having an encouraging, "Good," for his students, his incredible knowledge, patience, and devotion continue to awe and inspire me.

To **Ari Benbasat**, who always took time out of his busy schedule to give help, advice, witty sarcasm, and change for a dollar.

To **Kai-Yuh Hsiao**, who explained code to me.

To **Ari Adler** and **Jeff Hayashida**, who helped me build the scanning head.

To **Josh Strickon**, for leading me to Joe and the laser rangefinder.

To my fellow BatCave officemates: **Leila Hasan** for her music and **Mark Feldmeier** for his waffles.

And finally, to **my family**.

Table of contents

List of Figures

Chapter 1

Introduction

Human computer interaction, or HCI, is often achieved indirectly. You type on a keyboard, and letters appear on the screen. You move your mouse, and your movements are magnified and translated onto the screen. However, these interfaces make it obvious that there is still something that separates the user from the computer. Anything that is short of a touch screen, in fact, prohibits the user from interacting with the computer in a natural, hands-on manner. Even with a touch screen there are limitations, however. A user's movements are limited to the size of the touch screen itself, usually only the size of a standard 15" computer monitor, not accommodating the full capacity of human movement. To make a human-sized touch screen (or bigger), while not impossible, can be prohibitively expensive. If you wanted to use this touch screen with any other computer, you would need to transport this screen wherever you went. Being a touch screen, it is not likely that it would fold up into a compact, easily moveable size. What would be more desirable, then, would be an inexpensive, mobile system that could allow for multiple, large-scale user interaction. A touch screen, while able to track bare hand movement on its surface, is not easily adapted to bigger surfaces, or portable.

1.1 Existing Systems

While there exist other interfaces that allow for bare hand motions to be tracked atop large interactive surfaces, they all have their drawbacks. The Brain Opera's Gesture Wall [1], for example, used Electric Field Sensing [2] to track user movement, but required constant recalibration due to the physical differences of each user (thickness of shoes, etc.). Other examples of such Smart Walls involve video imaging and image processing [3] to track objects, some use infrared light and IR cameras [4, 5], while others use sonar to track active transponders [6-8]. Again, each has disadvantages, aside from being potentially expensive, they also require multiple sensing devices, complex processing to interpret the signals received, limit the user by posing restrictions (hand held targets, etc.), or do not possess the ability to track multiple targets. Laser rangefinders, on the other hand, while still expensive, have the advantage of being able to track motion with a single, unobtrusive device, needing less computation than image processing, and placing virtually no restrictions on the user, aside from staying within the wide sensing range of the device.

1.2 Laser Rangefinders, and Strickon's Prototype

Traditionally laser rangefinders have been used mostly for distance measurement survey [9], robotics [10], and military applications [11]. Being so functionally specific, they are usually either not suited to scan with too low a bandwidth, or too expensive and accurate for everyday needs. Joe Paradiso and Joshua Strickon [12], however, recently developed a low cost phase-measuring scanning laser rangefinder that had a range of several meters, centimeter resolution, and a 15-30 hertz scan rate, optimized to track the position of objects moving in a plane. Strickon built a working prototype for his Master of Engineering Thesis [13] for parts costing under $500.

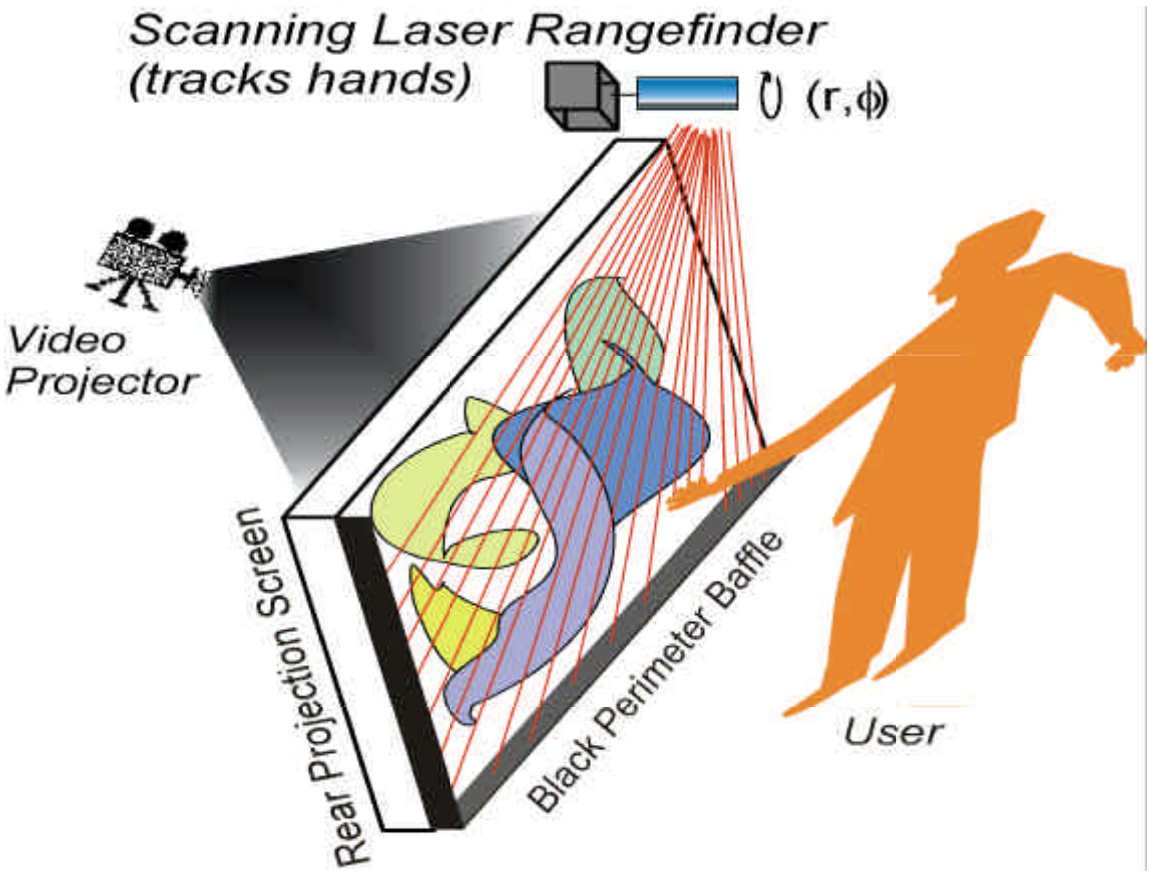


Figure 1-1: Laser Rangefinder Setup

By projecting an image onto any flat surface, and having the laser rangefinder measuring hand position above this surface from a corner, one could turn almost anything into a human sized touch screen (Figure 1-1). Not only is it an entirely natural interface and relatively inexpensive, it is also portable, consisting of a scanning head and a base unit. The scanning head contains a motor, optics, laser driver, photodiode and front-end electronics. The base unit houses the clock oscillator, demodulator, baseband electronics, microcomputer, and laser control. These will be described in more detail below.

1.2.1 How it Works

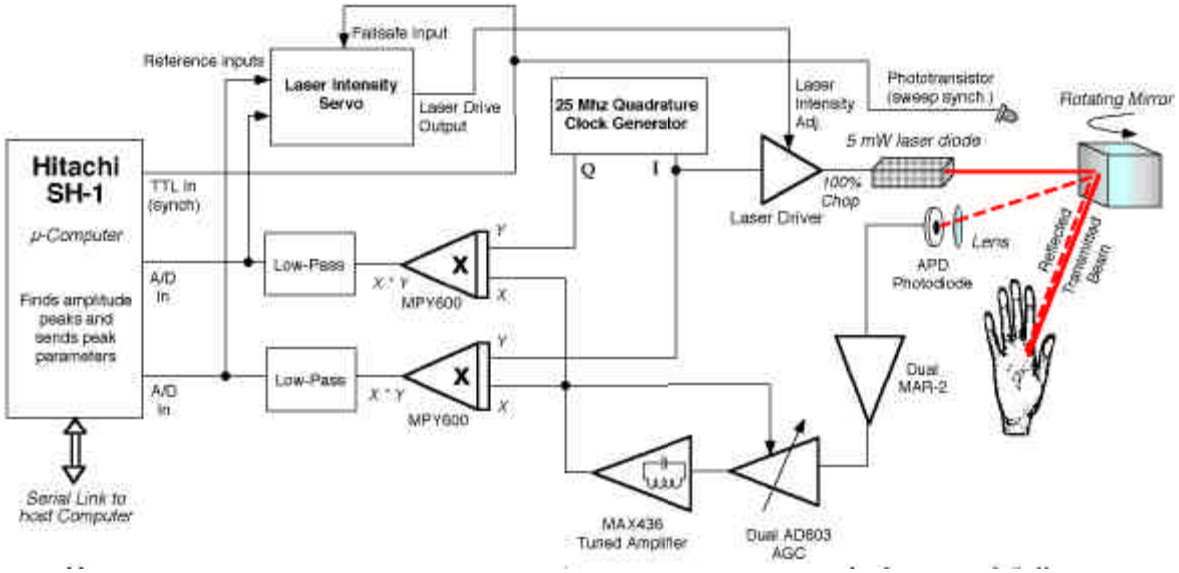


Figure 1-2: Block Diagram of Laser Rangefinder

A five milliwatt laser diode, as found in most laser pointers, is amplitude modulated using a 25 MHz signal. The modulation wavelength of the emitted laser is $\lambda_{mod} = c / f_{mod}$, where c is the speed of light ($3.0 * 10^8$ m/s) and f_{mod} is the 25 MHz modulation frequency, thus providing an effective wavelength of 12 meters. The useful range of the rangefinder, then, is half the modulation wavelength, or 6 meters; any further and phase-wrap occurs as the light travels to and then reflects back from the target. This modulated signal is pointed at a rotating mirror that reflects a "plane" of light, scanning at a rate of about 30 times a second, or 30 hertz. An avalanche photodiode biased at over 400 volts is used as a receiving sensor. The received signal is fed through a number of gain stages, then sent back to the base unit. In the base unit, the signal is fed into a pair of fast

multipliers, referenced by the direct clock signal and a 90 degree out of phase clock signal to obtain an in-phase and a quadrature demodulated signal. Peaks in these signals represent objects being tracked. These signals are then low pass filtered to select the baseband components, which are fed into a Hitachi SH-1 microprocessor [14] for analog to digital conversion, analysis and peak detection, and data transmission. The difference in phase of the amplitude-modulated emitted and received laser signal is accordingly determined by this quadrature pair (i.e., we make a continuous wave phase measurement) thus giving us the round-trip delay of the light, hence the distance from the laser *r*. The other polar coordinate, ?, is measured by timing the rotating mirror, as in a barcode scanner. A block diagram of the design is shown in Figure 1-2; and most portions are detailed in Strickon's thesis [13].

Figure 1-3 shows the laser rangefinder in action. The scanning laser rangefinder was first demonstrated inside the Media Lab using a simple drawing program as well as a musical application that created beats as the user moved multi-colored squares across the screen ("Rotating Cubes", Figure 1-3a). Pete Rice of the Opera of the Future Group subsequently developed Stretchable Music [15], which was used with the laser rangefinder and showcased at Siggraph 98 [16, 17], to create a melding of interactive music and graphics (Figure 1-3b). Users could move and stretch objects on the screen, creating music that was correspondingly moved and "stretched" along with their graphical representations.

a) Rotating Cubes '97 *b) LaserStretch '98* *c) LaserWho '99*

Figure 1-3: Laser Rangefinder Demonstrations

The laser rangefinder was also combined with Judith Donath's Visual Who [18] (from the Sociable Media Group) to create LaserWho. VisualWho is a visual database of people and their interests, affiliations, etc. By manipulating topics from a menu on the left of the screen, one can see people's names and infer their relationships with these topics, as the names spatially position according to how strongly they are associated to the topics. Combining this database with the scanning laser rangefinder allowed Visual Who to become fully interactive, along with music composed by Kai-Yuh Hsiao (of the Responsive Environments Group) that changed according to topic and how each topic was moved around (position, velocity, etc.). LaserWho was first publicly installed at Opera Totale 5 in Mestre, Italy (Figure 1-3c) in November of 1999, and again at Intel's Human Centered Product Innovation conference in January of 2000 with the old laser rangefinder (Figure 1-4). It was also showcased at Siggraph 2000 [19] with a new laser unit designed as part of this thesis. At Siggraph 2000, the LaserWho database consisted of Siggraph paper topics and their authors, as well as visitors who entered their information at the booth.

Figure 1-4: LaserWho at Intel Conference

1.2.2 Problems with the initial prototype

The initial prototype, while robust and capable of demonstrating the potential of our laser hand-tracker, had some problems that this thesis has addressed. First was the problem of sloppy mechanics. Being a prototype, the unit was not easy to adjust when the optics fell out of alignment, which happened often as the unit traveled from place to place. Also, it was difficult to obtain access to boards to test them, adjust them, or replace components and even more difficult to replace the boards themselves. The head unit itself was rather large and heavy, and had to be screwed onto a bulky mounting plate. The base unit was separate and consisted of three different boards encased within a rack mount unit. Among these boards one of them was a commercial evaluation board for the Hitachi SH-1, as well as one hand-wired board that had not been properly laid out yet.

While the rangefinder was capable of detecting multiple peaks from separate reflecting hands (and in fact, sent data for up to four such peaks to the host computer), this feature was not utilized to its full potential. Although Stretchable Music and LaserWho allowed for multiple objects to be moved around the screen, often one object would disappear behind another, as the closer object "shadowed" the further object when both lined up with the laser scan line.

This thesis will address these problems. In Chapter 2, the design of a new scanning head is detailed, as well as a compatible base electronics unit. Chapter 3 discusses the design of a new motherboard that combines the three previous base unit boards into one, reducing cost as well as size. It also describes the implementation of a new Analog Devices "microconverter" chip, which replaces the Hitachi SH-1 in both hardware and software. Chapter 4 presents a two-handed application that makes use of the laser rangefinder's multiple object tracking capabilities. The conclusions and suggestions for future work make up Chapter 5.

Chapter 2

New Head and Base Unit

2.1 Head Unit

For the next generation of the laser rangefinder, a new head was designed and built by Ari Adler, Jeff Hayashida, and myself. The new head was smaller and lighter than the previous head, and could stand by itself without having to be screwed onto a separate mount. The alignment of both the laser and the mirrors could be adjusted with separate tilt screws. Electronics were encased separately and more easily accessible, allowing for cleaner wiring. Three, four, and five pin XLR connectors replaced the original three, four, and four pin XLR connectors carrying power for the laser, camera, and motor (respectively), making it impossible to mix up the power cables by mistake and potentially harm the electronics (see Figure 2-3).

One of the most important improvements to the head unit was scan synchronization. Previously, two phototransistors were placed in the scan plane of the laser near the top and the bottom of the scan range of the old unit (see Figure 2-2), and synchronization occurred as the laser swept across these start/stop phototransistors. In the new head unit,

a custom shaft encoder mounted behind the mirror synchronized the scan sweep (Figure 2-1). This made the alignment process much easier, as synchronization was no longer dependent on the laser adjustments, while at the same time completely immunizing the rangefinder synchronization to background light. Start/stop, then, occurred as the slots in the encoder disk swept past a pair of encapsulated photodiode LED interruptor modules. The start/stop signals themselves could also be further adjusted electronically at the base unit board (detailed in the next section).

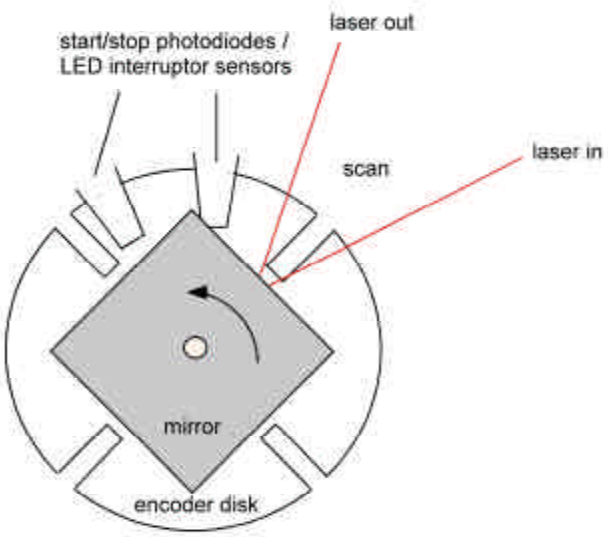


Figure 2-1: Shaft Encoder Mechanism with Start/Stop Photodiodes / LED Interruptors

It should also be noted that the new head used a DC motor with a belt drive, in comparison to the old unit which used a synchronous direct-drive motor (running near its slowest rate), thus allowing for smoother motor speed control.

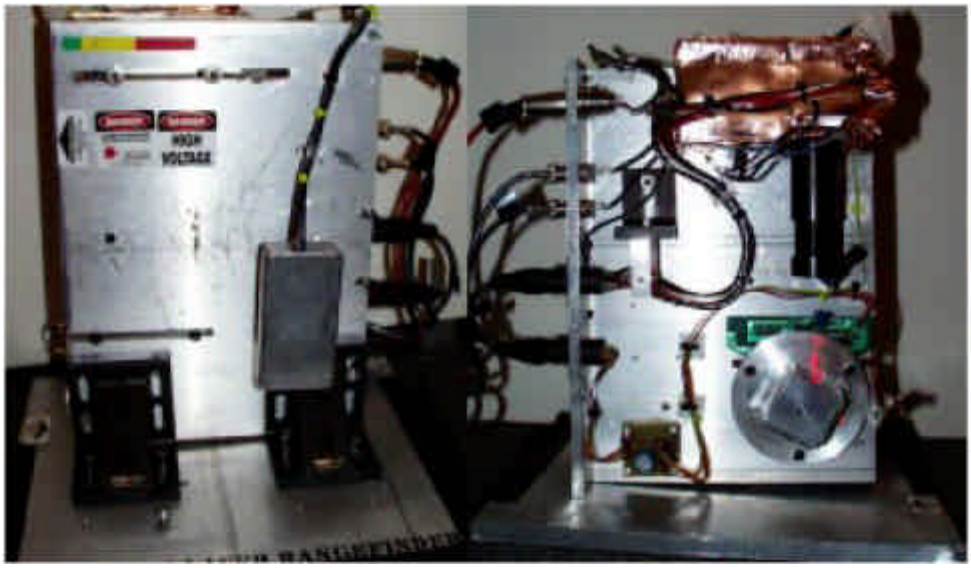


Figure 2-2: Old Laser Head: Back (Left) and Front (Right) Views

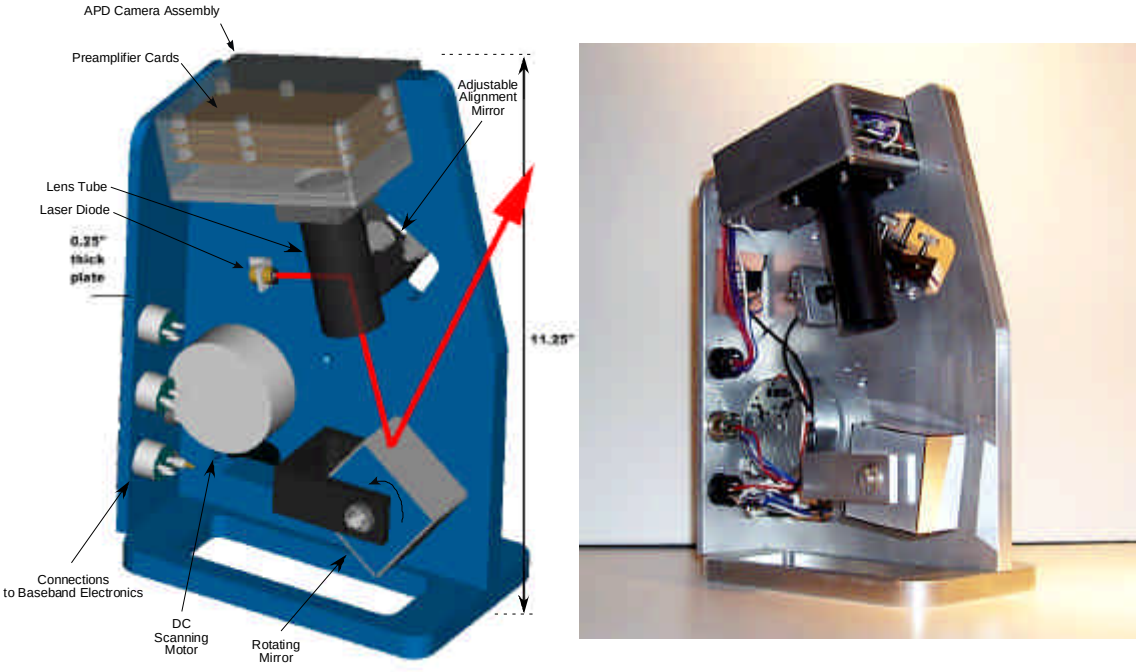


Figure 2-3: New Laser Head

2.2 Base Unit

With the changes to the head unit, the old base unit was no longer compatible with the new head, and a new base rack mount unit was designed and built (Figure 2-4). It contained the same demodulator and Hitachi SH-1 evaluation boards, but with a newly laid out board that controlled the "laser AGC", a darkside defeat switch, and a laser safety switch. The laser AGC, or active gain control, allows for the laser intensity to be adjusted through a feedback circuit, i.e., closer objects that create a stronger reflection signal cause the laser to dim, and vice versa. An additional AGC lowers the front-end gain, the laser AGC coming into play when the front-end AGC is saturated. When the darkside defeat is switched on, the laser will turn off when not scanning the area directly in front of the screen, preventing reflections from the inside of the head unit to be misread as data and allowing a clear DC baseline level to be acquired. The laser safety switch turns off the laser when the motor is not running, so that the laser will never be pointing at a still mirror and reflect directly into anybody's eye for a significant interval of time (hence our system can be classified as eye-safe [20]).

This new board, called the "extra laser board", also supplied adjustable power to both the motor and the laser. A voltage regulator circuit provided adjustable motor power ranging from about 5 to 12 volts, which accordingly changed motor speed. Laser power came from a potentiometer connected in series with resistors and +12 volts to provide about 3-5 volts. This voltage was buffered by an emitter follower before driving the laser. It controlled the maximum brightness of the laser, and with that the strength of the signal. It could be feedback-controlled by the strength of the detected signal (the laser AGC). Also located on the board were potentiometers used to adjust the start/stop signals (both trigger thresholds and time delay), as well as diagnostic LED outputs for serial receive and transmit (Figure A-2).

The newer redesigned laser rangefinder was much smaller and sleeker, having even more operational range and sensitivity than the old rangefinder, due to the better degree of

optical alignment that was possible. It was used in the LaserWho installation at Siggraph 2000 in New Orleans (mentioned earlier), as well as British Telecomm's Global Challenge Technology Showcase at the Harbour View Hotel, Boston in October of 2000.

The base unit (as seen in Figure 2-4), however, still consisted of three separate boards: an 8"x6" demodulation board, a 5"x5" microcontroller eval board, and a 6"x2.5" laser control board. The demodulation and laser control board shared a separate power supply from the eval board, and wiring between the three boards was complex.

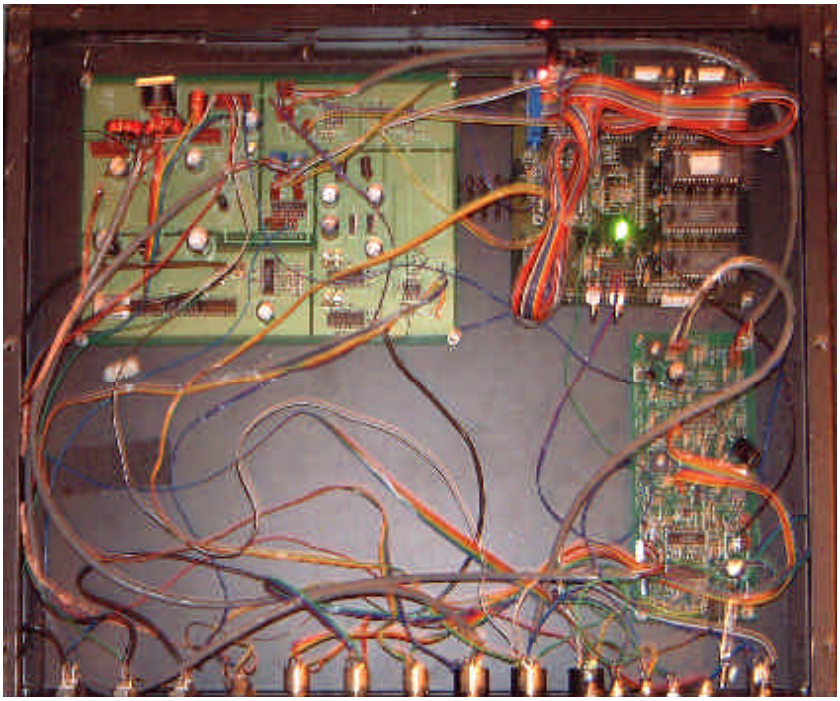


Figure 2-4: Redesigned Base Unit and Face Plate

Wherever we went with the device, we kept getting requests from people who wanted their own laser rangefinder for similar applications. Research colleagues, Media Lab sponsors, companies, small startups, museums, and even artists took a great interest in having one to use for their own purposes. We could not, of course accommodate these requests, only having two laser rangefinders; the previous prototype and the one recently built. The redesigned head unit could potentially be contracted out and built, since it had a simpler design, and used only a few simple boards. However, the base unit, with its three complicated, separate boards, including a rather costly Hitachi SH-1 evaluation board as well as lots of complex intraboard wiring, required more streamlining before it could be economically reproduced.

Chapter 3

New Redesigned Base Unit

A new motherboard for the base unit was designed, incorporating the three boards in the previous iteration of the base unit into a single board.

In order to insure a proper square wave as a phase reference (deviations can induce quadrature errors), the 50 megahertz (MHz) master clock for the laser was replaced with a 100 MHz crystal oscillator. To obtain a modulation frequency of 25 MHz, the signal from the 100 MHz clock was fed into the clock input of a 74VHC393, a 4-bit high speed CMOS binary counter, that divided the signal into a 50 MHz clock signal, insuring a 50% duty cycle (not guaranteed at the output of clock chips). This signal was fed into the original quadrature clocking circuitry, which used a pair of D flip fops to produce an in-phase (I) and a quadrature (Q) clock reference at 25 MHz. I and Q were fed separately into two Burr-Brown MPY600 fast multipliers, to be demodulated by the received signal, then low-pass filtered, creating two basebanded signals. These signals were then conditioned by a voltage-controlled amplifier (VCA), allowing the gain of both to be simultaneously adjusted, and a 5-volt rail-to-rail op-amp, then fed in as inputs to the A/D converter of the embedded microcontroller (Figure A-3). Much of the rest of the

demodulation circuitry (Figure A-1) and laser control circuitry remained the same, with only the board layout changing. The SH-1 microcontroller was replaced with Analog Devices ADμC812 microconverter, an 8-bit microprocessor with a 12-bit A/D converter, as detailed in the next section. The microconverter portion of the layout consisted of the ADμC812, external data memory as well as a socket for optional external program memory, and serial communications circuitry. Surface mount resistors and capacitors were used instead of through-hole parts to conserve board space. All of these subsystems were laid out to fit on one 6”x12” board, running from a single power source. (Figure 3-1, A-4 & A-5)

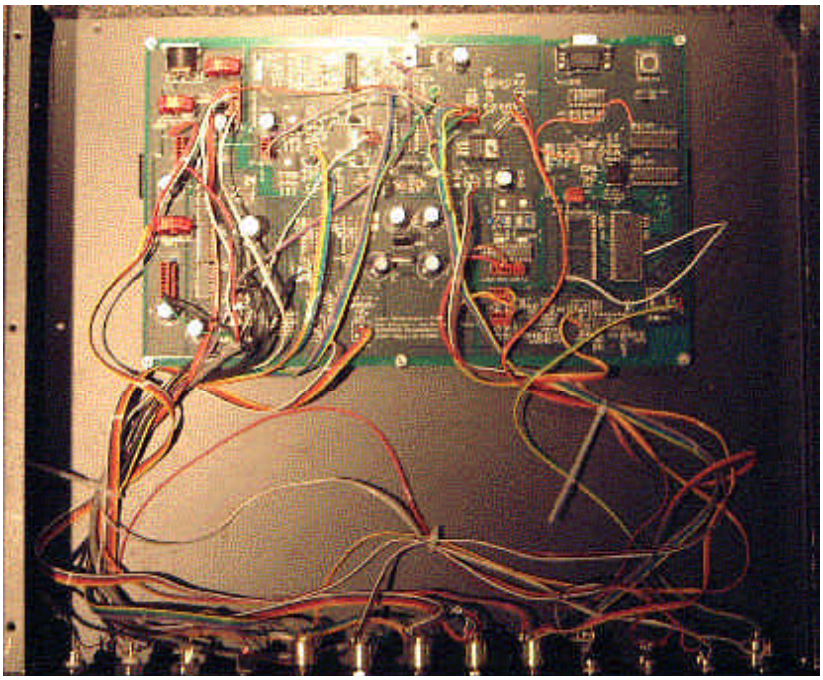


Figure 3-1: New Motherboard and Base Unit

Signals that had to be routed from board to board were now directly routed through traces, greatly reducing the amount of external wiring required (compare Figs. 2-4 and 3-1). The cost of the microconverter, along with all the chips needed to support it, reduced

the cost of the computing circuitry by almost an order of magnitude. Requiring only one board to be manufactured, as opposed to two custom boards plus the SH-1 evaluation board, further reduced cost. With the base unit now encasing only one board, it is easy to imagine the motherboard being attached directly to the laser head itself, and the scanning laser rangefinder becoming a single, portable unit. Of course, the board could easily be scaled down even further, using all surface mount components, while placing connectors and LEDS on the edge of the board. With these adjustments, external wiring would be eliminated completely, while shrinking the board down to roughly half or less of its current size.

3.1 Microconverter Circuitry and Software

3.1.1 Hardware

The microconverter that replaced the Hitachi SH-1 was the Analog Devices ADμC812, built around an 8051 microcontroller core. Its features include a 12-bit, 5 microsecond (μs), 8-channel analog to digital converter (ADC), 8K bytes of EE/FLASH program memory, 640 bytes of EE/FLASH data memory, 256 bytes of data RAM, as well as three on-chip timers. While less powerful than the Hitachi SH-1, it was much cheaper and seemed to suit our needs, while having a higher resolution ADC (12-bit vs. 10-bit) [21].

The setup of the microconverter was based roughly on the ADμC812 evaluation board [22], as well as the processor architecture of Ari Benbasat's Inertial Measurement Units [23], with at first an 11.0592 MHz then a 16 MHz oscillator for a clock, for reasons discussed later. Two SN54HC573AJ D-type octal latches were used to latch the addresses and data for the external memory. An HM62256 32k word x 8-bit SRAM chip was used for external data memory. A socket for an NM27C512Q90 64k word x 8-bit CMOS EEPROM was also available for external program memory. As it turns out, the code for the microcontroller was smaller than 8K bytes, and therefore external program memory was not needed, hence the socket could be removed in future layouts. A

MAX232 served as the serial line driver, connected between the microconverter's serial output and a standard RS-232 serial port. Channel 1 (in-phase) and Channel 2 (quadrature) outputs from the demodulator were fed as inputs into the ADC, while start and stop signals were wired to extra input pins. The /EA pin was pulled down to ground to allow for external memory access, and two pushbutton switches were used to control the program and reset pins, allowing the microconverter program to be loaded in-situ (Figure A-3).

Since the original laser rangefinder's channel outputs were conditioned by the VCA (see above) and offset adjustments to be between 0 and 5 volts, the external voltage reference pin was wired to 5V (with the appropriate bypass capacitor) to override the ADμC812's internal AD reference voltage of 2.56V in order to prevent ADC saturation. Alternatively, the demodulation gain could have been reduced so that the channel outputs did not exceed 2.56 volts, but we opted to match the maximum ADC input with the final-stage op-amp rails.

Running at slower clock speeds (11.0592 and 16.0 MHz) with 12 cycles per instruction (roughly 1 and 1.33 MHz) compared to the 20 MHz (single cycle per instruction) Hitachi microcontroller, as well as having relatively very little on-chip data memory, posed some difficulties in porting the code over to the AdμC812. The code had to be optimized so that the laser could continue to scan at its original speed, while at the same time taking in and analyzing as many or more samples per scan.

3.1.2 Software

Code originally written by Strickon and Hsiao was directly ported over to the new microconverter in C using Keil's µVision compiler and assembler [22].

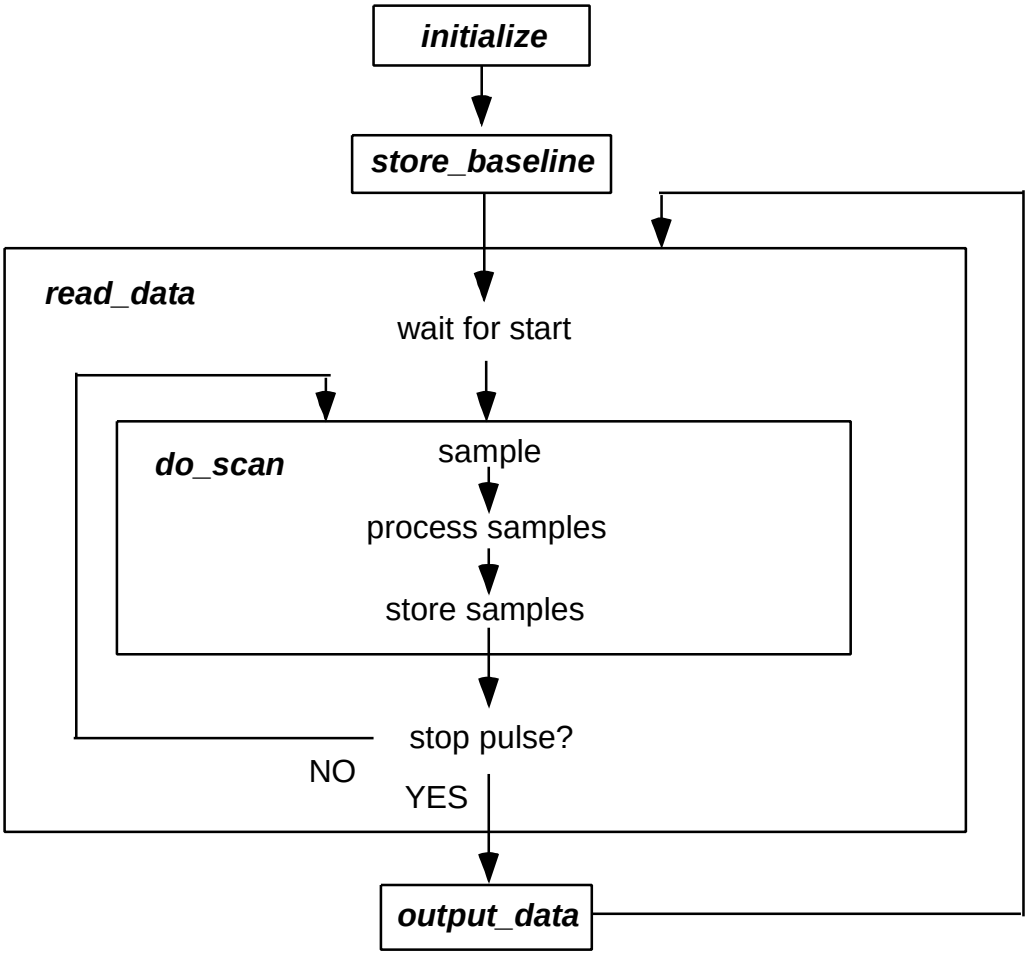


Figure 3-2: Old Code Flowchart

In Strickon and Hsiao's original code (Figure 3-2), upon pressing reset, the program would initialize all its variables, set up its timers as well as serial communication (in *initialize*). *Store_baseline* would then be called. In *store_baseline*, within a scan cycle (a scan cycle defined as the time between when a start pulse goes high and the next time a stop pulse goes high, see Figure 3-3), the two data channels would be sampled, added onto previous values and stored. This is repeated for 8 separate scan cycles, and an average baseline taken. Next the *read_data* function is called. This is the main bulk of the program, which calls *do_scan* repeatedly for the duration of a scan cycle. Within *do_scan*, the two channels are sampled, then compared to a threshold value to see if the values constitute a valid reflection peak (the laser is scanned against a matte-black baffle (Figure 1-1) hence hands are detected by their reflection peaks (Figure 3-4)). If they do, then they are added to previous same-peak values and stored. At the end of the scan cycle, *output_data* is called to transmit the signal data in packet form (detailed below), and *read_data* is called again.

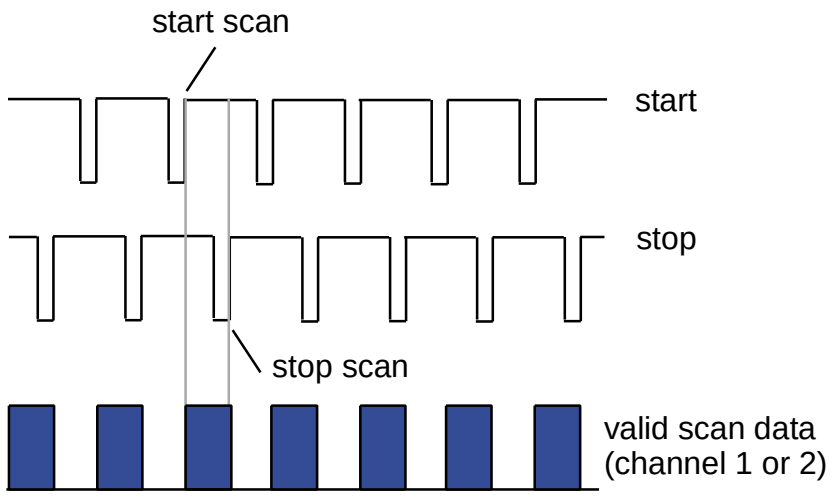


Figure 3-3: A scan cycle is defined as the time between a rising edge of the start pulse and the next rising edge of the stop pulse - these edges are adjusted to sandwich the sensitive scan region.

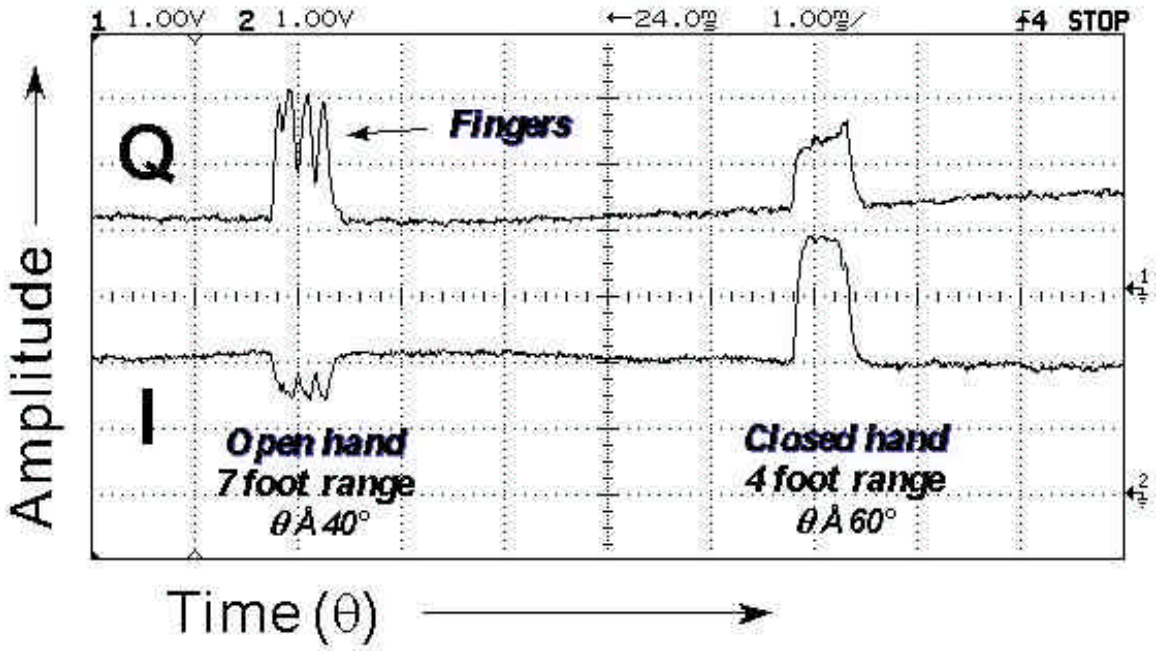


Figure 3-4: Tracked Objects and their Corresponding Peaks in I and Q

Running the microprocessor at a slower clock rate as well as having to constantly access external data memory slowed down the rate of the output data significantly. There were several attempts made at correcting this. First, in the original code, every sample had to go through a series of calculations and then stored before the next sample could be taken. This was because Strickon had a faster, more powerful processor that was able to do calculations on the fly and still keep a high scan rate.

The Analog Devices microconverter, however, was unable to execute this code fast enough. Although it only takes 5µs to convert each sample, the ADµC812 does not have a setting that allows it to sample two or more channels at once, so sampling two channels together required a function written in code. After each sample was taken, it had to be stored in external memory, as well as run through a series of calculations, before the next sample set was taken. As a rough estimate, converting two samples takes over 10µs, since we have to switch which channel we are reading in code. Each external data memory access also takes about 5µs, but only one byte is written at a time. To write a 12-bit result, it thus takes 10µs for each sample to be written. As such, it takes over 30µs for

each sample to be converted, and written to external data memory. Then the samples have to be retrieved from external memory again, compared, processed, and peak results stored back again in memory. This took too much time, and as a result the microconverter was only able to take about 40 samples within a scan cycle, using the old code scheme.

Two things were done to fix this. First, a faster master clock was used. Originally the microconverter ran with a master clock frequency of 11.0592 MHz. I replaced the clock with a 16MHz crystal oscillator (maximum clock rate for the AdμC812), which provided for faster sampling, as well as faster overall instruction cycles. I optimized the code wherever I could, utilizing the microconverter's fast on chip data memory for variables that were used the most often.

The most important change to the code, however, was to its structure (Figure 3-5). The code still ran in a similar way to the previous code. Upon reset, initialize was called, and then *store_baseline*. In *store_baseline*, however, instead of sampling, calculating, storing, then sampling again, during a scan cycle the two channels were sampled and stored into external data memory continuously, with calculating done after the scan cycle was over. A similar approach was taken for *read_data*. In *read_data*, instead of calling *do_scan* during the scan cycles, which sampled, calculated, and stored data, *read_data* simply sampled and stored continuously during the scan cycle. After a scan cycle was over, it called a new function, called *calc_scan*. *Calc_scan* did all the calculations that *do_scan* did, except it used data that had already been sampled and stored in external memory. In this way I was able to increase the number of samples taken per cycle to over 280, well surpassing the number needed for our applications. (refer to Appendix B for code).

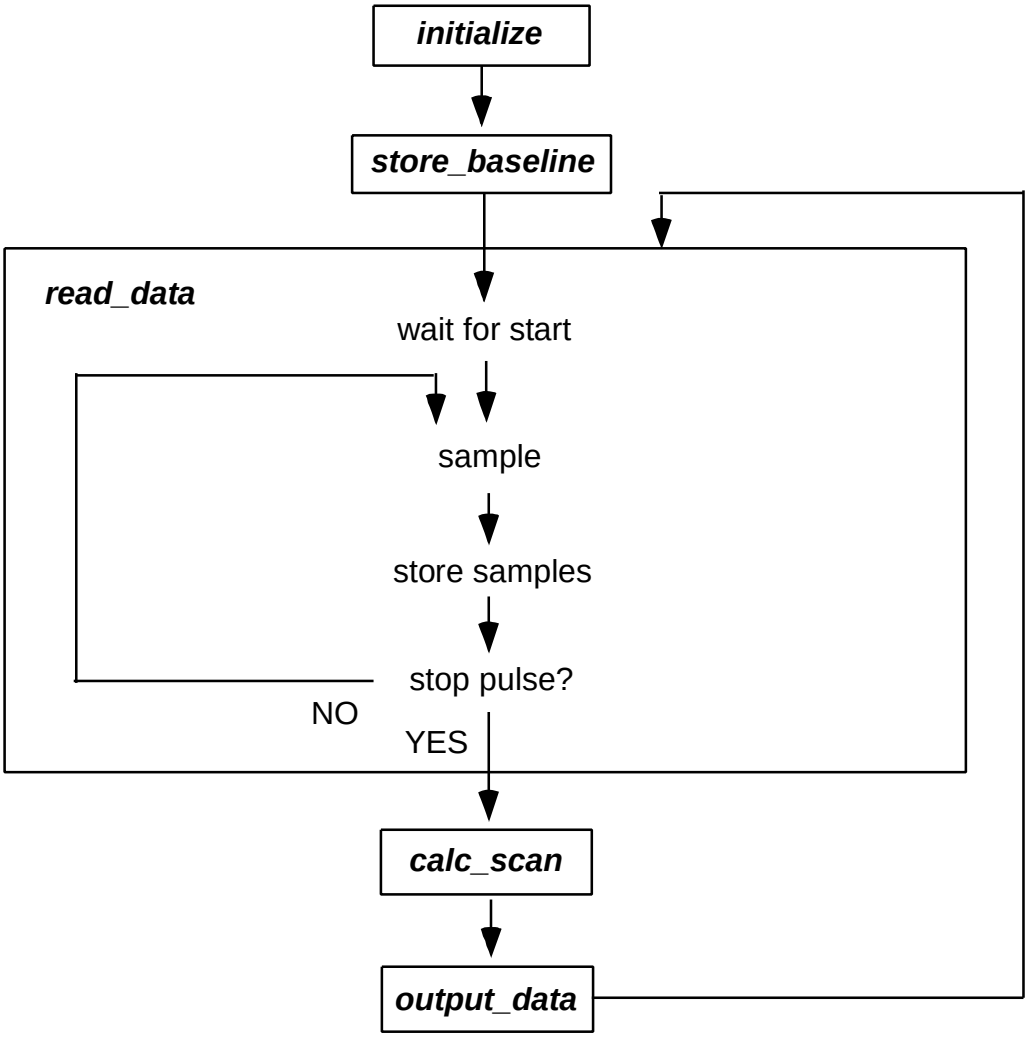


Figure 3-5: New Code Flowchart

However, in changing the microconverter clock frequency, the serial baud rate was affected. The 16-bit timer used for serial timeout was an 8-bit reloading timer that incremented with each machine cycle (or 12 oscillator periods). With an 11.0592 MHz clock, each machine cycle period was about 1.085 µs, allowing us to set the timer to reload with 0xFD. The timer would therefore reload every (0x10000 – 0xFD00)x1.085 µs, or about once every 833 µs. This is the time that it should take to send an 8-bit value out (not including stop or start bits). Therefore, dividing 833 µs by 8, and taking the inverse, we get a serial transfer rate of 9600 bits per second (bps). This baud rate is

doubled through a special function register on the ADμC812, giving us a final transfer rate of 19.2 kbps. However, with a 16 MHz clock (or 0.75 μs instruction cycle), the serial timeout timer, reloading only its higher 8 bits, could not be set at a high enough resolution to allow a 19.2 kbps serial baud rate, or multiple thereof. As a result, the timer had to be reloaded with 0xFA, thereby providing a (close to) 14.4 kbps baud rate (higher multiples were not supported, to be explained later), much slower than the old laser rangefinder's 38.4 kbps rate.

With the optimized code but slower serial communication rate, the result was a 7.5 Hertz update rate taking data from only one out of four scans, as opposed to the original 15 Hertz update rate, or one out of two scans. The tradeoff was having over 280 samples per cycle, as compared to the about 220 samples per cycle of the original device.

To try to improve the scan rate, problems with the code were pinpointed and addressed. First, in the old code, data was sent serially every scan cycle, even if there were no reflection peaks found in the signal. The packet was always a fixed length of 77 bytes: 4 header bytes, followed by number of peaks (1 byte) and 18 data bytes per peak, enough to accommodate up to four peaks. If there were no peaks, this packet consisted of the header bytes, while the remainder of the packet was padded with zeros. If there were less than four peaks, the remainder of the packet would also be padded with zeros. Most of the time, there is only one peak in the signal, requiring only about a fourth of the bytes to be sent, and thus only a fourth of the time it takes to send the entire packet. To send only relevant peak data, however, required significant code changes on the PC interface side to deal with dynamic packet sizes, and was not implemented at the time of this thesis.

Another problem in the code happened when it sent out data. In the original code, data was sent out all at once at the end of the scan. With a more powerful processor, this is possible without slowing down the scan rate much, since scanning and calculating take less time. However with the microconverter, calculating the data took too much time, and as a result, sending the data all out at once at the end of the scan cycle resulted in a

slower scan rate. To try to remedy this, two different ways of using the microconverter's interrupts were attempted. Both versions put data that was to be sent directly into a FIFO buffer as soon as it was ready. The buffer was a user defined-size *char* array, maintained by an indexing pointer, a pointer that kept track of the oldest data, as well as a variable to keep track of the total number of bytes in the buffer, and a *nodata* flag that was set to 1 when there was no data in the buffer to send.

The first version of this approach utilized the microconverter's internal timer 2, setting it as a reloading timer. Whenever the timer overflowed, an interrupt would be triggered, checking to see if there was any data to be sent out. If there was, then the oldest data within the FIFO buffer would be written to the serial buffer register, and then transmitted out. This saves time because we never have to wait for the serial port to finish transmitting before we can write to the serial buffer register. The second version of this approach utilized the actual transmit interrupt. Whenever a transmission was completed, the serial transmit interrupt would be triggered, causing the next byte of data in the FIFO to be written to the serial buffer register and transmitted.

This approach encountered problems with the code, however, and performance was not reliable. Data was sent sporadically, and often times the data was corrupt. I attempted to debug the code, with no success. Instead, I identified several ways in which the microconverter or the code was unfit in utilizing interrupts to increase the scan rate, leading to the observed failures.

To send the same type of packet that was mentioned earlier (77 bytes, regardless of the number of peaks, padded with zeros), the interrupt method wouldn't have saved much time unless you had more than one peak. With the interrupt method, data is sent out over a set of interrupts in the *calc_scan* function, which takes the most amount of time. However, not until *calc_scan* has calculated the data from over 200 samples does it know how many peaks the data has. If you had no peaks, then the entire 77 bytes minus the header bytes would have to be written with zeros and sent, but with no time to send them,

as *calc_scan* will no longer be called until the next scan cycle. If the 77 bytes have not been sent out yet, then they could be overwritten in the next cycle, and the data would be corrupted. The same is true for any number of peaks less than 4, since the data has to be padded at the end of the *calc_scan* function. Therefore our approach was not suitable, largely because of the structure of the code, leaving no time to send actual data once it had been calculated. Due to time constraints, this method was abandoned, but revisited below.

As mentioned before, the number of peaks is unknown until after the *calc_scan* function, but it was sent right after the header in the original code. In the interrupt method, data was sent during the *calc_scan* function, and the number of peaks had to be the last thing you sent out. Of course, this could be corrected on the PC interface side of the code. You could also eliminate the number of peaks variable entirely, since you can infer the number of peaks from the data sent. Since a header byte is attached only to the beginning of each set of scan cycle data, if the data continues past a certain length without a new header, you know that data about a new peak is being sent.

3.1.3 Increasing the Output Rate

To increase the output rate from 7.5 Hertz to 15 or even 30 Hertz, we must somehow minimize the time that the *calc_scan* function takes to analyze the close to 300 samples that are acquired. Since the old laser rangefinder only took a little over 200 samples, we can easily limit the number of samples that the rangefinder takes, by setting a limit in the code. This not only cuts down on sampling time, but also allows the code to call the *calc_scan* function as soon as the sample limit is reached. This would cut down on the number of samples that *calc_scan* has to analyze, further reducing the amount of time it takes to run through each iteration of the program per cycle.

As of the time this thesis was finished, the serial baud rate of the microcontroller was only set at 14.4 kbps. This was due to the fact that the timer did not support a serial

connection rate of 19.2 kbps with a 16 MHz clock as explained earlier. Also during testing, the serial programs that I used did not support higher multiples of 14.4 kbps speeds. However, since serial speeds are set in the software on both the microprocessor side as well as the PC interface side, higher multiples of 14.4 kbps speeds can be used, with appropriate coding.

As mentioned above, the data packet is incredibly inefficient, often being stuffed with zeros. The code can be rewritten on both the microprocessor side as well as the PC interface side so that the data packet is one that changes size dynamically, according to how much information is in the single. Data for a single peak should be sent in a smaller packet than data for multiple peaks.

The interrupt method can also be retried. Instead of having a single buffer, we can use a double-buffering approach, where the data for one scan is stored in one buffer, to be sent out during the next scan cycle, as the other buffer is being written with new data.

However the biggest obstacle in increasing the scan rate is reducing the time the *calc_scan* function takes, as this is the bottleneck in the program. This can involve doing several things to make *calc_scan* faster, mostly involving reducing the time taken to access external data memory.

First, since we didn't utilize the 640 bytes of on-chip flash data memory (which is much more quickly accessible than the external data RAM), we could conceivably store the baseline (only taken once when the rangefinder is first powered) there, instead of in external data memory as it is now. However, with the number of samples per scan that the microconverter is converting, this would be impossible. Even though the baseline is only taken once, with over 280 samples for each of two channels, along with baseline times (for every two baseline samples taken from two separate channels, an 8-bit indexing time is also stored), this would require about 2.24 kbytes of data memory. To get around this, we could greatly reduce the baseline time resolution (as it tends to move

slowly), and only take every four or five samples. Another possibility is to only store an 8-bit baseline, e.g., bit-shifting the ADC output, storing it, and then bit-shifting it back to get a 12-bit baseline padded with zeros to compare with scanning data. Yet another option is to use a single, average baseline number, and only store deviant peaks or shifts (from reflections in the baffle) in the baseline.

Another solution is more brute force; e.g. to substitute another embedded microprocessor with more computing power. We could go back to the SH family, building around the processor itself instead of its (expensive) evaluation board, or research others. For example, Analog Device's ADMC328 [24] is a 20 MIPS, 28-pin DSP that can do calculations while sampling, with a 512 x 16-bit data memory RAM. Another example is Cygnal's 8051 family [25], with 20 MIPS or greater instruction rate and up to 2.3 kbytes of on-chip RAM. Based on the 8051 microprocessor core, the code written for the ADμC812 could be ported easily to this device.

Given today's processor speeds, all the processing could be done on the host computer, requiring the microconverter to only convert samples and then send the data serially. This would cut down drastically on the processing time of the microconverter, since it can convert one channel and send data while it is converting the other channel. Furthermore, external data memory access will no longer be necessary, further improving speed. With these improvements, the scan output rate could very well be increased to 30 times per second, or even more if the motor speed were correspondingly raised. As an estimation, for 250 samples per scan cycle, each sample consisting of two 12-bit values (in-phase and quadrature) and an 8-bit timer value, 8000 bits need to be sent every scan cycle. For an output rate that matches the laser scan rate of 30 Hz, a serial transfer rate of 240 kbps is required, comparable to data rates commonly encountered with digital audio.

Chapter 4

Two-handed Application

4.1 Why Create a Two-handed Application?

The whole idea behind our laser rangefinder is to encourage human interaction with computers. With a personal computer, because of the small display and intimate interface, it is difficult for multiple users to interact with the computer at the same time, hence the term “personal computer”. However with our laser rangefinder installation, interaction now becomes human-sized, and more than one person can easily fit into the interaction space. In fact, when one person is interacting with the system, a crowd often draws as more and more people try to track their own hands. When LaserWho was showcased at Siggraph2000, more than one person would often be at the booth, simultaneously trying to interact with the topics on the screen. However, when most people tried to do this, their efforts were thwarted as soon as someone who was closer to the scanning unit effectively “blocked” other hands with their “shadow”(see Figure 4-1). Even when only a single user was interacting with LaserWho with two hands [26], control was ambiguous. Because of this, applications for the rangefinder generally were made to only support single object control. If multiple users attempted to manipulate

objects on the projected screen, these rangefinder applications would decide, sometimes seemingly arbitrarily, which object had “control”, and only this object would respond to user movement. As a result control would switch occasionally, from user to user, providing for a counterintuitive interface. However, the rangefinder is very sensitive, and can detect separate peaks (hence multiple hands) only centimeters away from each other. To not allow full multiple user interaction would not be utilizing the rangefinder to its full potential. One solution, of course, is to use two or more laser rangefinders. While this would eliminate all problems of occlusion, this method is expensive and complicates the needed hardware. Thus a software solution was pursued, and a two-handed application was written in C++.

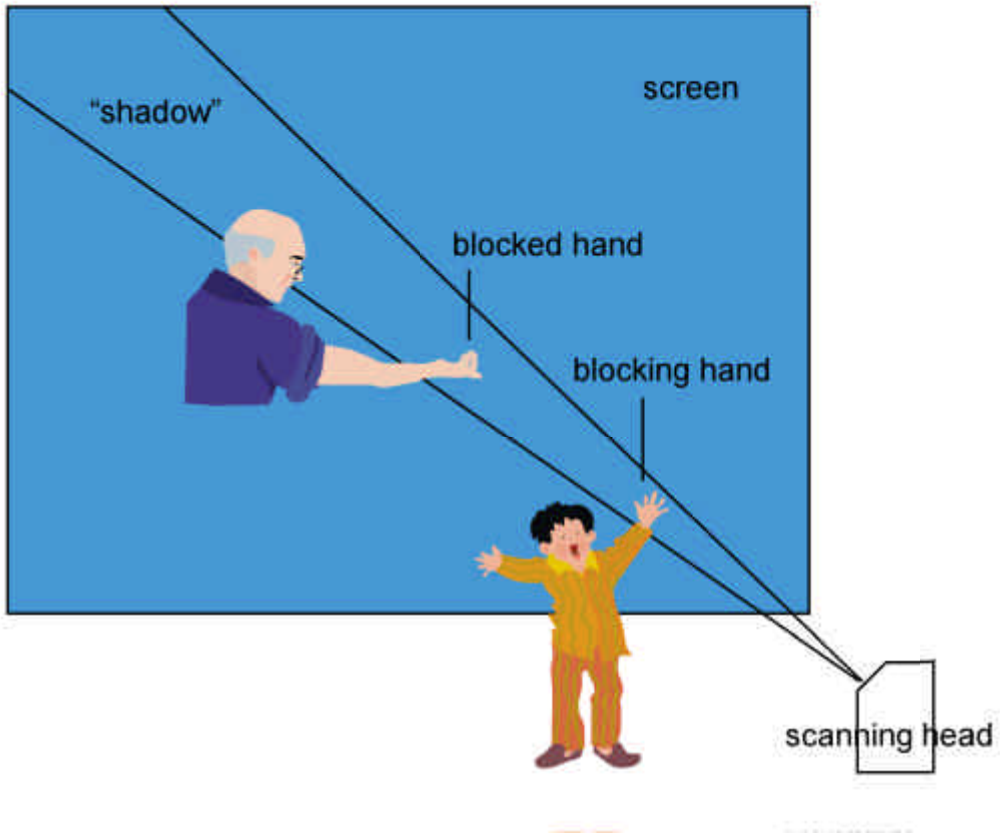


Figure 4-1: Shadowing

There were interesting obstacles to overcome when programming a true two-handed application for the laser rangefinder. First, how would one go about representing a hand

that was shadowed by another? Secondly, when do you decide whether the hand remains in the shadow of the closer hand, or is already out of the plane of detection? And lastly, the point of origin of the screen is not generally the point of origin of the rangefinder itself, as the scanning unit is always placed above or below the screen.

4.2 Code

The basic idea of the code for the two-handed application was to extrapolate a signal's trajectory when it became blocked by another signal. The code was built upon existing C++ PC laser interface code, originally written by Josh Strickon and Kai-Yuh Hsiao. This original laser interface code took in the data stream of peaks coming from the laser base unit, and first calculated their polar coordinates (which the device intrinsically wants to produce), then transformed them to rectangular coordinates via a third order polynomial. For demonstration and test purposes, the data points were then represented as multi-colored cubes in a graphics window that could be resized by the user. The code also had a built-in calibration function, in which calibration data was written to a file and run through a Matlab function written by Joe Paradiso to create a calibration coefficient file. This file was then used in the translation of polar coordinates to rectangular coordinates (details in [12, 13]).

Since the original laser interface code deals with the transformation of *r* and ? from the rangefinder to x and y coordinates on the actual screen display, we decided to deal directly with the *r* and ? coordinates before they were transformed. In this way the problem of offset was avoided, since extrapolated *r* and ? coordinates would be translated like regular data points.

The value of *r* was calculated by taking the arctangent of the ratio of (In-phase sum / Quadrature sum); here, the more negative *r* was, the closer the target was to the rangefinder. ? was represented by the midpoint of the peak (as ? is determined by the scan time of the rotating mirror, ? = (start time + finish time) / 2)). In reality, these are

"pseudo" polar coordinates; *r* is actually a number in radians and ? is a number in clock cycles. As such, we had to be careful of the wrap-around from 2p to 0. In the original code, if *r* was positive, then 2p was subtracted from the *r* value. However, in testing out the new base unit, it was found that *r* values furthest away from the head unit were about –4. Closer to the laser, the *r* values increased negatively, then there was a jump in *r* values changing from –6 to 0, about two feet away from the laser scanning head, after which *r* values continued to increase negatively again to about –2. This was due to the actual *r* values becoming positive starting two feet away from the laser, but with 2p subtracted because they were positive values. To prevent this "*r* wrap", an offset was put into the code that could be set by the user. This offset was automatically added to each *r* calculated so that every *r* would be positive, then made negative by the original *if* statement that subtracted 2p as mentioned above. In the current version of the code, this offset was set to 3, allowing *r* values to range from about –1 to –5 (far to near).

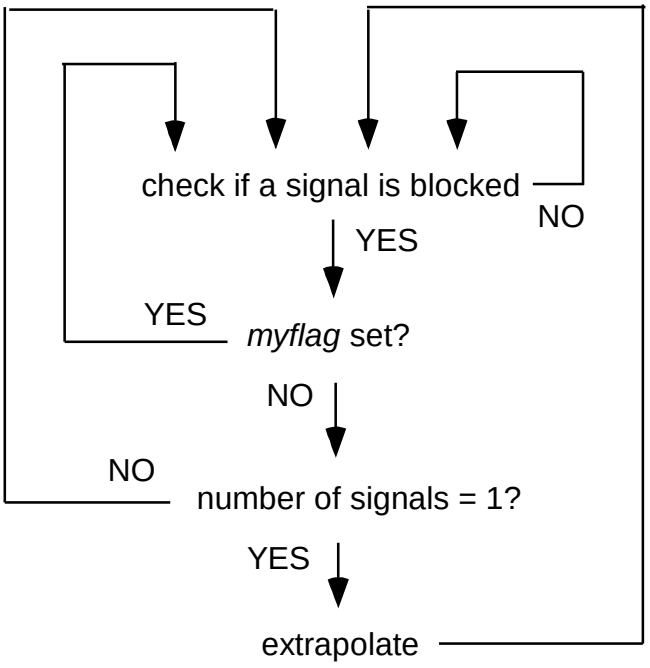


Figure 4-2: Basic Flowchart for Two-handed Extrapolation Logic

A basic flowchart of the structure of the code is shown in Figure 4-2 above, and the code itself is located in Appendix C. The code had to keep track of state in order to extrapolate from previous values, as well as determine which hand was blocking which. To do this, state-keeping arrays of *r* and ? with sizes defined by the user were created (*SAMPLES*). While the rangefinder can handle more than two objects at once, this experimental two-handed application was written to support only two objects, so only two state-keeping arrays each of *r* and ? were kept. With each data point collected, these arrays were updated so that they contained the most recent samples. The values within the arrays were simply shifted over and the newest point stored in the vacant slot. The first signal was always written into the first array, while the second was always written to the second array. However, when two signals became one as both hands came into shadowing alignment, only one array was written with data and the other with zeros (to be overwritten later). If the second signal was blocking the first signal, and only one signal was seen by the rangefinder, the second signal would thus be seen as the first and only signal by the rangefinder, hence the first array would be written with data from the second signal. Therefore, it was necessary to keep track of which signal was blocking which. This was done by comparing *r* values. As mentioned previously, the more negative the *r*, the closer the signal is to the rangefinder. Thus by finding which signal is closest to the rangefinder, we know which signal is the blocking signal and which signal is being blocked (see Figure 4-1). If the second signal was closer to the rangefinder before the two signals became one, then a flag was set so that even if the rangefinder only saw the one signal, the data would still be written to the correct (second) array, and the first data array written with zeros.

Next the code checked along ? to see if a signal was about to block another signal. When two signals come close to one another, the end time of one peak comes close to the start time of another. If the end time of one signal plus a certain time threshold (set by the user) was greater or equal to the start time of another signal, then a *blocked* flag would be set. Since a signal was not actually being blocked yet, a flag (*myflag*) was set so that no extrapolation (see below) would occur in the next part of the code. Another flag was also

set indicating which signal came before which. This is used later in determining which direction a signal was coming from when extrapolating the trajectory for a blocked signal.

If the *blocked* flag was set, and *myflag* was also set, then the code continued without extrapolation, resetting *myflag* but still keeping the *blocked* flag set. The code then checked to see whether two signals had become one signal. If not, then the *blocked* flag was set to zero and the code continued without extrapolation. If there was now only one signal, however, and the *blocked* flag was set, then the blocked signal was extrapolated.

In extrapolation, first the number of signals was incremented to two instead of one. Then the *r* and ? velocities were calculated to extrapolate the trajectories of the blocked point forward based on its velocity before being occluded. The *r* velocity was calculated by taking the difference between the oldest and newest samples within the blocked signal array, and dividing by the number of samples in the state-keeping arrays (*SAMPLES)*.

The ? velocity was calculated slightly differently, by taking the difference between the start and finish times of the blocking signals (not from the previous blocked signal values) and dividing by the number of samples in the array. The code had to be aware, however, which way the blocked signal had been coming, so that the ? velocity was correspondingly either positive or negative. To do this, two things were checked. First, which signal was blocking which; if signal 1 was closer, then signal 2 was being blocked. Next, the flag that indicated which signal had come earlier was checked. If signal 1 was the blocking signal, and signal 1 had come earlier than signal 2, then signal 2's extrapolated ? velocity was positive, or counterclockwise. Likewise, if signal 1 was the blocking signal, and signal 2 had come earlier, then signal 2's extrapolated ? velocity was negative, or clockwise.

Once the *r* and ? velocities were calculated, they were added to the second-to-the-most-recent *r* and ? samples, and written into the newest slot in their respective state-keeping arrays. There was also a *countdown* integer, which was set to the number of elements in

the array. Once *countdown* reached zero, the blocked signal should have appeared on the other side of the blocking signal, and was kept there until a new signal was detected, whereupon *countdown* would be reset and both hands were tracked normally. This was done by writing the blocked signal's newest array slot with the most recent value written into the array.

In Figure 4-3 on the next page, the two-handed application is demonstrated. In the last screen, the cube looks like it is offset from the outer hand, but that was in fact due to lag and poor calibration. The application worked, although it can be fine-tuned by the methods described above.

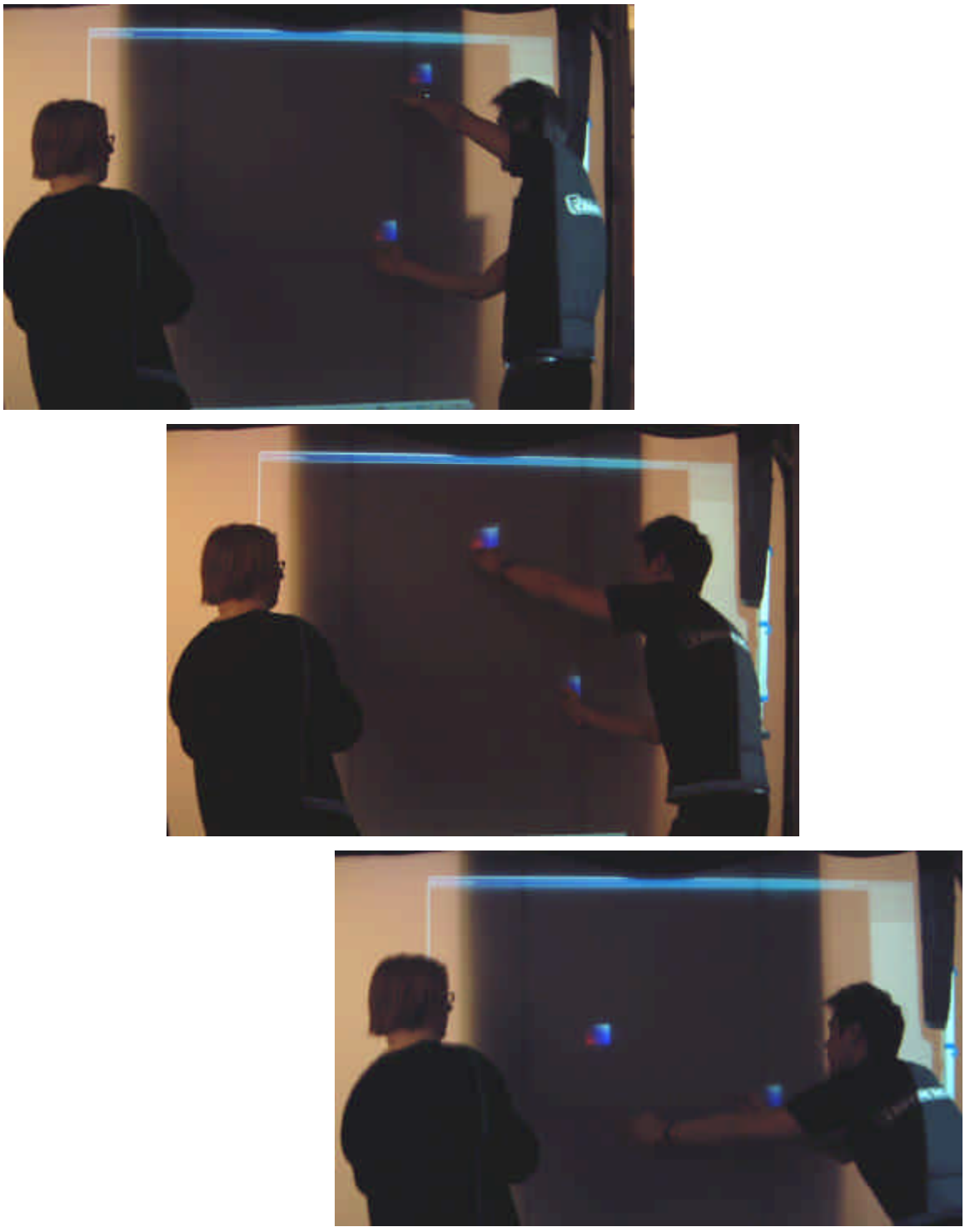


Figure 4-3: Two-handed application in action as the outer hand sweeps through shadow of inner hand

4.3 Improving the code

There are several simple ways to improve this code, some of which have already been written into the code. However, most of these changes also require some testing to find what works best. One example is changing the number of state-keeping samples that the code keeps track of (*SAMPLES*). At the time of this thesis the number of samples was set to 5, but it can easily be changed by the user in a define statement, to a value that requires as little computation as necessary but also provides good extrapolation properties. Another value that can easily be adjusted is the time threshold, used to determine when one object is about to block or be blocked by another object. This value can also be optimized through more testing of the code.

Minimum *r* velocity was a parameter that was included in the code but commented out. This is easy to change, as more testing of the code can yield a minimum velocity threshold. This minimum *r* velocity is used to extrapolate the position of the blocked signal if the extrapolated signal *r* (or ?) velocity is smaller than the minimum velocity. Currently, ? velocity is calculated using the start and finish times of the blocking signal, and not by actually extrapolating the blocked signal's true velocity (mentioned above). Since the blocked signal is guaranteed to appear at the opposite side of the blocking signal's shadow with this method of calculating ? velocity, a minimum velocity is not required. However, if the actual ? velocity of the blocked signal is later added into the code, then a minimum velocity would probably become necessary.

Another possible improvement (also already written into the code) is to lock the blocking signal. While this would restrict movement of the blocking signal until the end of extrapolation, it would prevent the blocking signal from drifting in between the two signals. This occurs as the two thinner peaks become a single, wider peak, and the rangefinder sums up the total in-phase and quadrature values for the wider peak, resulting in a different *r* value.

A last improvement that would be simple to add would be a timeout on how long a blocked signal stays on one side of the other signal. In the current version of the code, the blocked signal stays on the screen indefinitely once the *countdown* variable has reached zero. The only way to move the object from this position is if the rangefinder senses a second peak, whereupon the plotted object moves to this new position. This new position, of course, is often right next to the extrapolated object, as the blocked signal moves out of the shadow of the blocking signal. However, if a user moves his hand out of the scanning plane while being blocked, then the program will keep a second object on the screen until a second peak appears again. This is easy to deal with, by testing the program and deciding how long *countdown* should wait for a second peak to appear until it automatically removes the second object, thereby assuming the blocked hand was removed while it was being blocked.

Chapter 5

Conclusion

This thesis traced the process of redesigning the laser rangefinder, making it cheaper and smaller. First, a new scanning head was built that was smaller, lighter, and easier to align than the original prototype [13]. Then two base units were built, one encasing three separate boards, and one that held a single motherboard with a new and simpler microprocessor. Code was ported over to the new microprocessor, and was restructured so that enough samples were taken every scan cycle. Finally, a two-handed application was written to demonstrate the multi-hand tracking capabilities of the laser rangefinder.

5.1 Future Work

Several improvements to the current system, both in hardware and software, were mentioned above. This included shrinking down the size of the motherboard even further, using all surface mount components, with connectors and LED's mounted on the

edge of the board. With such a small board and no external wiring, the base unit could be eliminated entirely and the motherboard mounted onto the scanning laser unit itself, providing for a single mobile unit. Software improvements, detailed above, included increasing the scan data output rate, either through more efficient code or replacing the microprocessor with a faster, more powerful one.

Other hardware improvements to the general system would be the implementation of a heterodyne circuit, creating an intermediate frequency (IF) stage between the received signal and the basebanded signal. Taking the received signal and mixing it to an intermediate frequency would allow another amplifying stage or AGC to be put in between the IF stage and the baseband stage. This would allow for further tuning and noise reduction, with the possibility of eliminating the high gain but expensive avalanche photodiode and replacing it with a cheaper PIN photodiode. On the software side, the code could be written to detect peaks in range, rather than amplitude. While this would eliminate the need for a baffle around the screen (used to prevent phase-wrap of far-away reflected signals), it would require a more powerful microprocessor for signal analysis, either calculating the arctangent explicitly or using a lookup table.

Improvements to the two-handed application were also mentioned (and a large part of them already coded in). However, since the two-handed application was more a proof-of-concept more than anything, it could be adapted into a truly multi-handed application, not limited to just two hands. This would require more processing, however, as more states need to be tracked. Also a more complicated estimator can be developed to better extrapolate target trajectories. Other tricks with graphics can also be written in to encourage the user to move away from shadowing alignment.

While trivial, it is important to mention that during the course of this thesis, the laser pointer diode on the scanning head died, possibly due to overdriving. The resistor on the laser driver board can be replaced with a higher value to decrease current through the laser, to prevent overdriving the laser in the future.

References

1. Paradiso, J.A., *The Brain Opera Technology: New Instruments and Gesture Sensors for Musical Interaction and Performance.* Journal of New Music Research, 1999. **28**: p. 130-149.

2. Paradiso, J. and N. Gershenfeld, *Musical Applications of Electric Field Sensing.* Computer Music Journal, 1997. **21**(3): p. 69-89.

3. Wren, C.R., et al., *Perceptive Spaces for Performance and Entertainment: Untethered Interaction using Computer Vision and Audition.* Applied Artificial Intelligence, 1997. **11**(4): p. 267-284.

4. Matsushita, N. and J. Rekimoto, *Holowall: Designing a Finger, Hand, Body, and Object Sensitive Wall.* Proceedings of UIST '97, 1997.

5. See http://www.csl.sony.co.jp/person/rekimoto/holowall/.

6. See http://www.virtual-ink.com.

7. Salamone, S., *Kantek Fingers a Better Mouse.* Byte Magazine, 1996.

8. See http://www.worklink.net/products/ringmouse.html.

9. Rueger, J.M., *Electronic distance Measurement: An Introduction*. Fourth ed. 1996, New York: Springer-Verlag.

10. Everett, H.R., *Sensors for Mobile Robots: Theory and Application*. 1995, Wellesley, MA: A K Peters, Ltd.

11. Urban, E.C., *The Information Warrior*. IEEE Spectrum, 1995. **32**(11): p. 66-70.

12. Paradiso, J. and J. Strickon, *Tracking hands above large interactive surfaces with a low-cost scanning laser rangefinder.* ACM CHI '98 Extended Abstracts, 1998: p. 231-232.

13. Strickon, J., *Design and HCI Applications of a Low-Cost Scanning Laser Rangefinder*, M.S. Thesis, *MIT Media Lab*. 1999, MIT.

14. See http://semiconductor.hitachi.com/superh/.

15. Rice, P.W., *Stretchable Music: A graphically rich, interactive composition system*, M.S. Thesis, *MIT Media Lab*. 1998, MIT.

16. Strickon, J., P. Rice, and J. Paradiso, *Stretchable Music with Laser Rangefinder*. SIGGRAPH 98 Conference Abstracts and Applications, 1998: p. 123.

17. Phillips-Mahoney, D., *Laser Technology Gives Video Walls a Hand.* Computer Graphics World, 1998. **21**(10): p. 17-18.

18. Donath, J.S., *Visual Who: Animating the affinities and activities of an electronic community.* ACM Multimedia 95 - Electronic Proceedings, 1995: p. 99-107.

19. Donath, J., et al. *LaserWho*. in *SIGGRAPH 2000 Conference Abstracts and Applications*. 2000: ACM Press: p. 86.

20. Ready, J.F., *Industrial Applications of Lasers*. 1997, New York: Academic Press.

21. See http://products.analog.com/products/info.asp?product=ADuC812.

22. See http://www.analog.com/microconverter/.

23. Benbasat, A.Y., *An Inertial Measurement Unit for User Interfaces*, M.S. Thesis, *Media Arts And Sciences*, MIT Media Laboratory. 2000, MIT.

24. See http://products.analog.com/products/info.asp?product=ADMC328.

25. See http://www.cygnal.com.

26. Buxton, W. and B.A. Myers, *A study in two-handed input.* Proceedings of CHI '86, 1986: p. 321-326.

Appendix A

Schematics and PCB Layouts

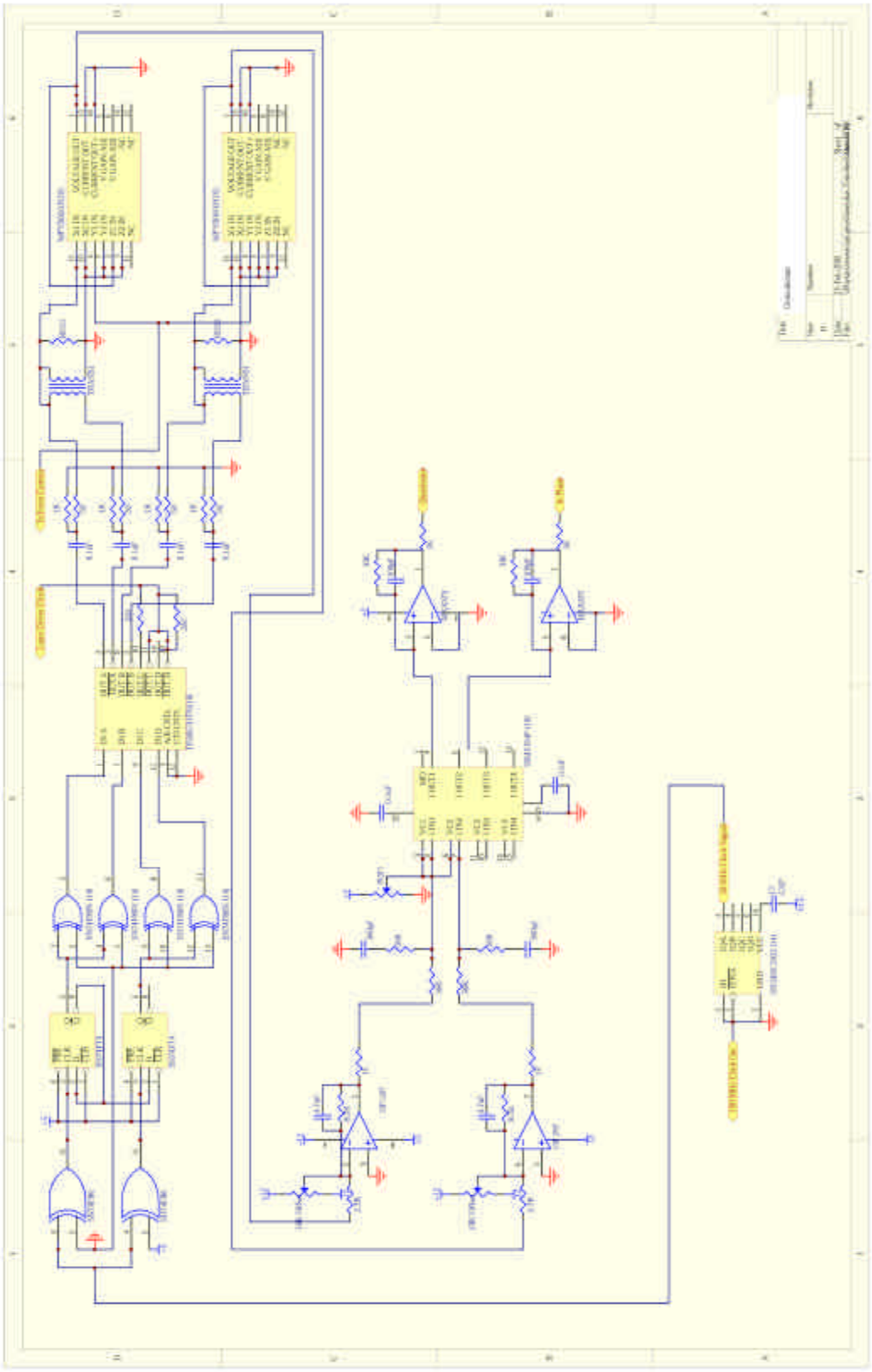
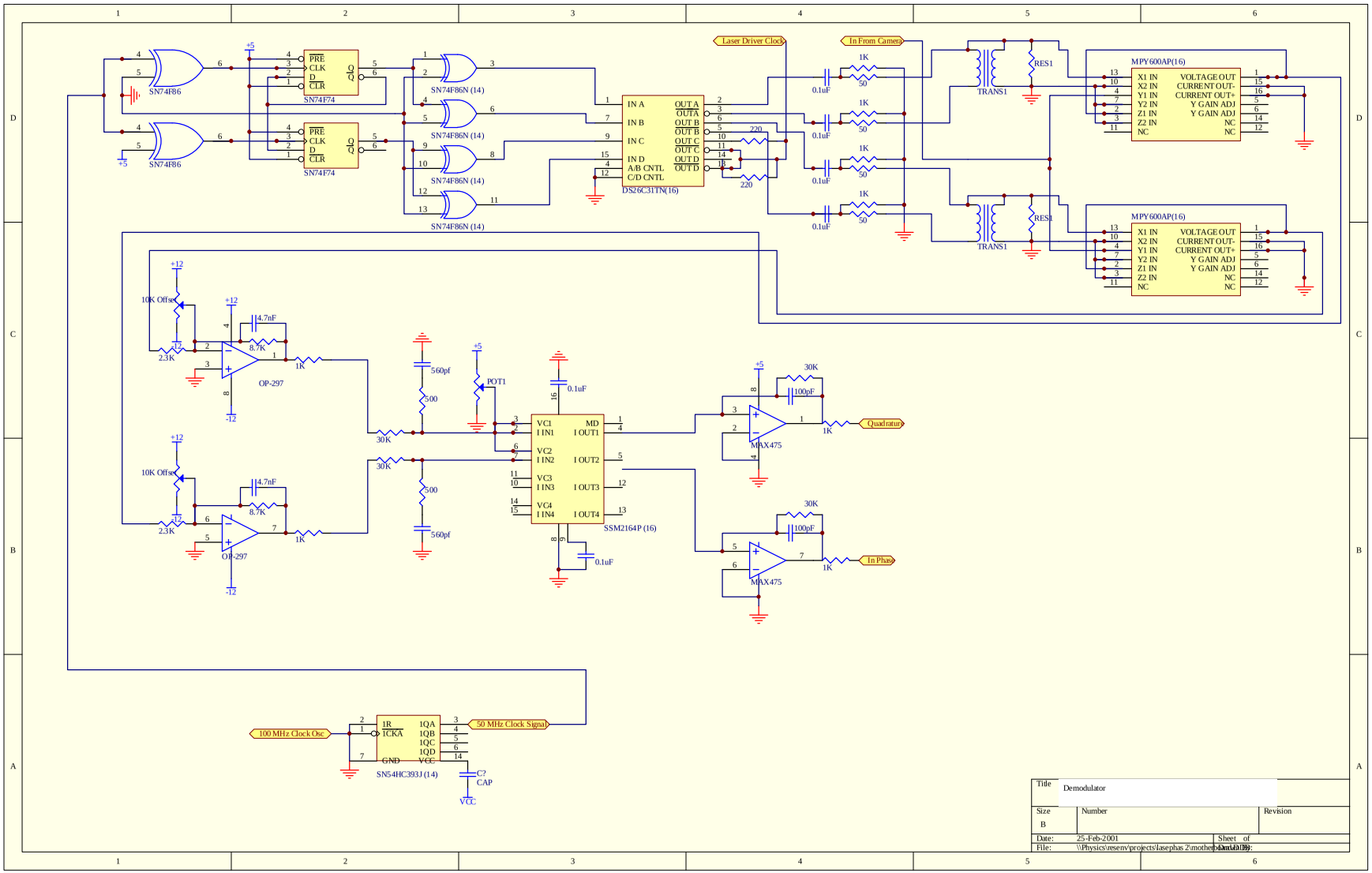


Figure A-1: Demodulator Schematic


Laser Driver Clock
In From Camera
SN74F86
SN74F74
SN74F86N (14)
DS26C31TN(16)
0.1uF
1K
50
220
TRANS1
RES1
MPY600AP(16)
X1 IN
X2 IN
Y1 IN
Y2 IN
Z1 IN
Z2 IN
NC
VOLTAGE OUT
CURRENT OUT-
CURRENT OUT+
Y GAIN ADJ
10K Offset
2.3K
4.7nF
8.7K
OP-297
560pf
500
30K
POT1
SSM2164P (16)
100pF
MAX475
Quadratur
In Phase
100 MHz Clock Osc
50 MHz Clock Signal
SN54HC393J (14)
C2
CAP
VCC
Title
Demodulator
Size
B
Number
Revision
Date:
25-Feb-2001
Sheet of
File:
\\Physics\resenv\projects\lasephas 2\motherboard.ddb
Drawn By:

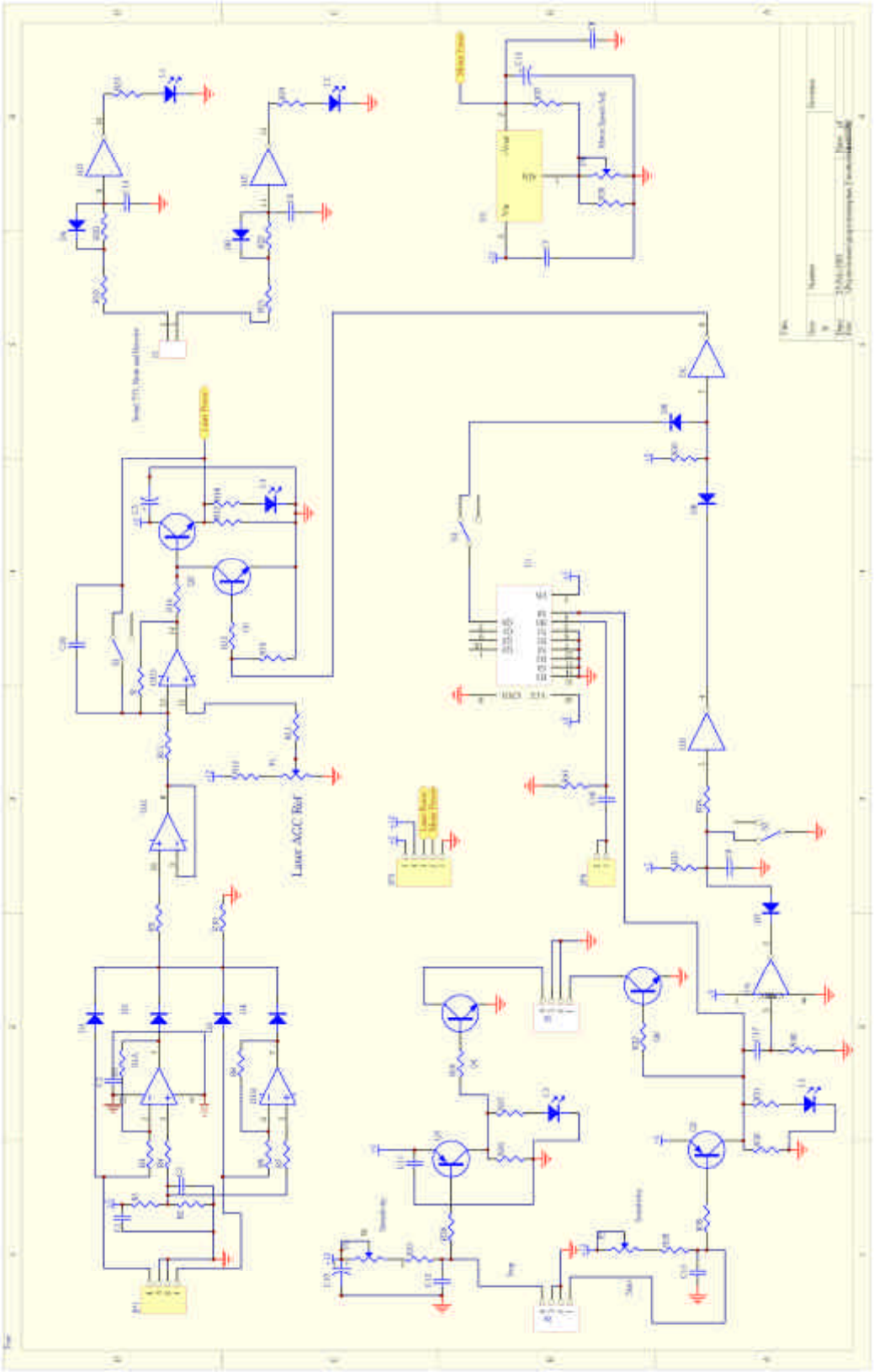
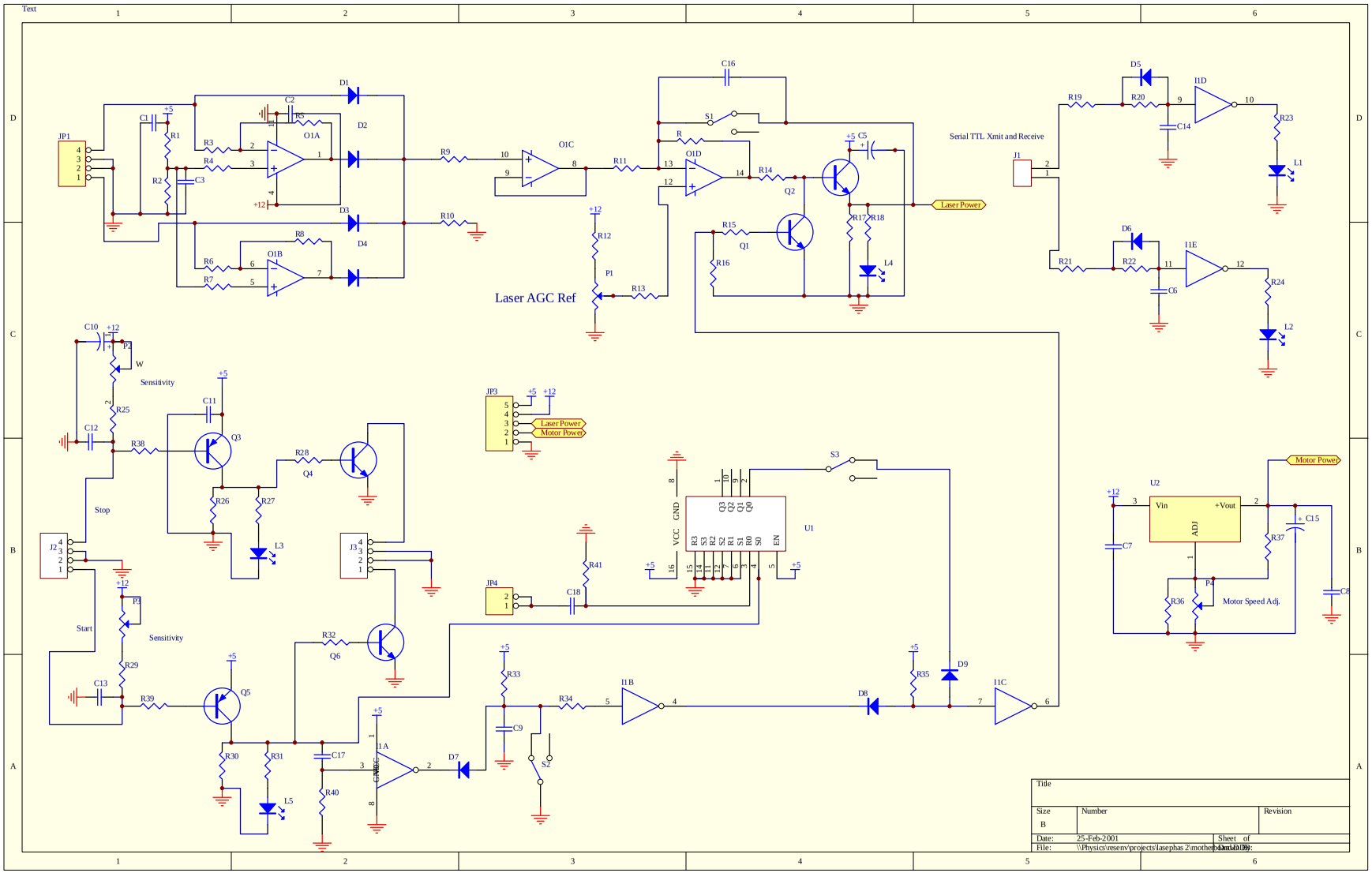


Figure A-2: “Extra Laser” Circuitry Schematic


Text
JP1
C1
R1
R2
C3
R3
R4
C2
R5
O1A
D1
D2
D3
D4
R8
O1B
R6
R7
R9
R10
O1C
C16
S1
R
O1D
R11
R12
P1
R13
Laser AGC Ref
R14
Q2
R15
Q1
R16
C5
R17
R18
L4
Laser Power
Serial TTL Xmit and Receive
J1
D5
I1D
R19
R20
C14
R23
L1
D6
I1E
R21
R22
C6
R24
L2
C10
P2
W
Sensitivity
R25
C12
R38
C11
Q3
R28
Q4
R26
R27
L3
Stop
J2
J3
JP3
Laser Power
Motor Power
JP4
C18
R41
U1
S3
GND
VCC
EN
U2
Vin
+Vout
ADJ
C7
C15
R37
P4
Motor Speed Adj.
R36
C8
Start
P3
Sensitivity
R29
C13
R39
Q5
R32
Q6
R33
R34
I1B
I1C
D8
D9
R35
C9
S2
I1A
D7
R30
R31
C17
R40
L5
Title
Size
B
Number
Revision
Date:
25-Feb-2001
Sheet of
File:
\\Physics\resenv\projects\lasephas 2\motherboard.ddb
Drawn By:

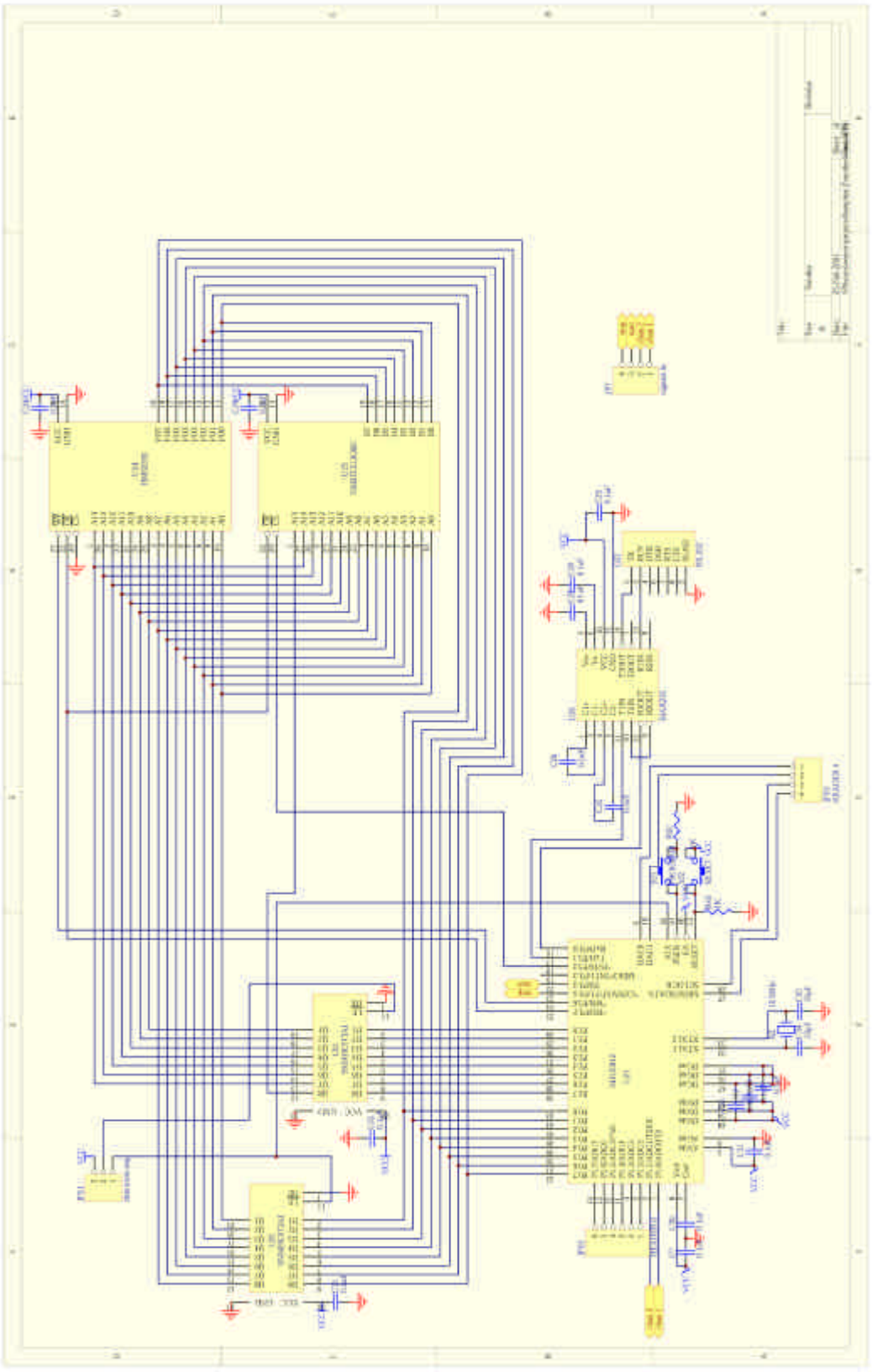
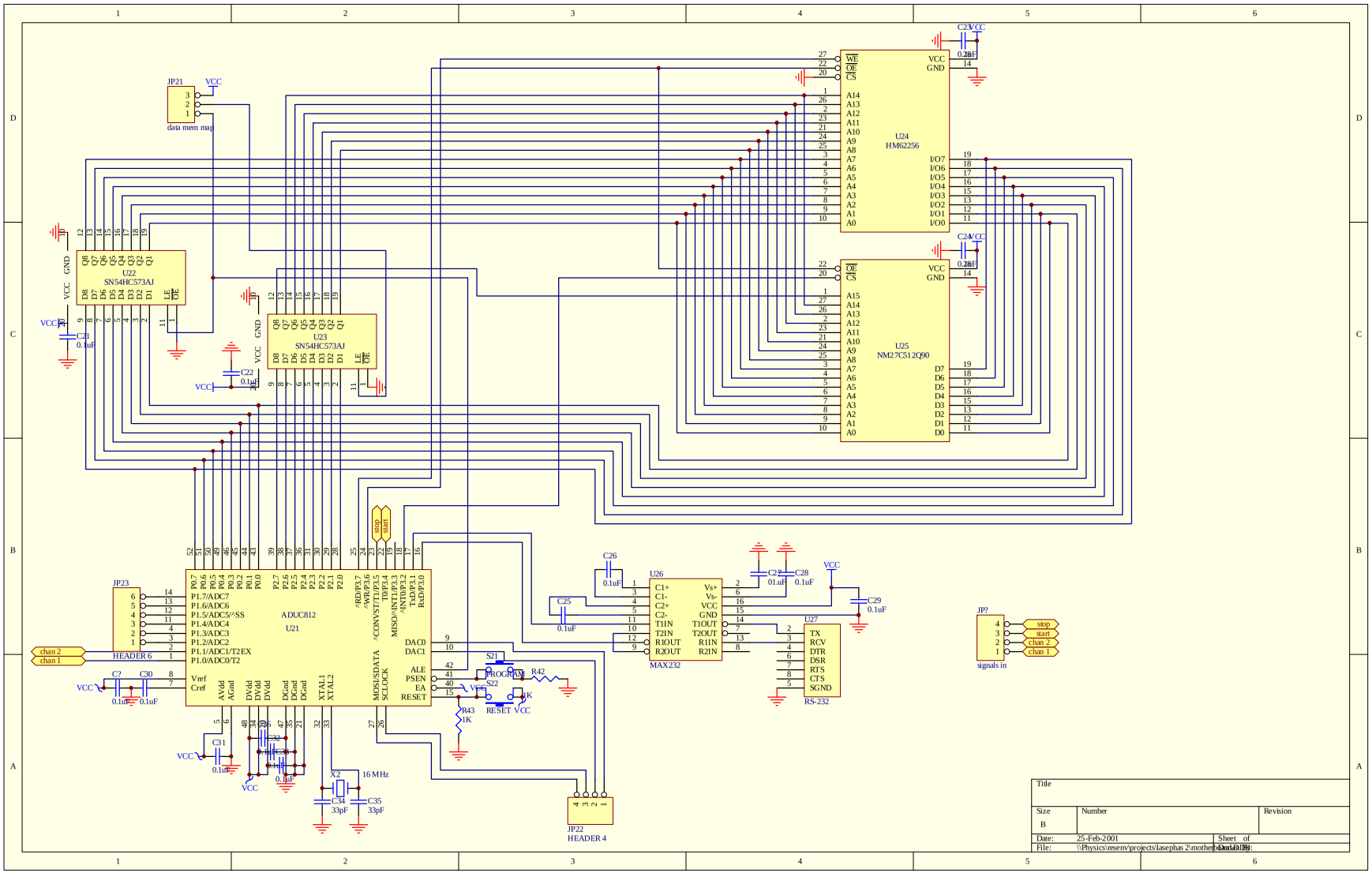


Figure A-3: Microconverter Circuitry Schematic


JP21
data mem map
U22
SN54HC573AJ
U23
SN54HC573AJ
U24
HM62256
U25
NM27C512Q90
ADUC812
U21
U26
MAX232
U27
RS-232
JP23
HEADER 6
JP22
HEADER 4
JP?
signals in
chan 2
chan 1
stop
start
PROGRAM
RESET
X2
16 MHz
C34
33pF
C35
33pF
R42
R43
1K
VCC
Title
Size
B
Number
Revision
Date:
25-Feb-2001
Sheet of
File:

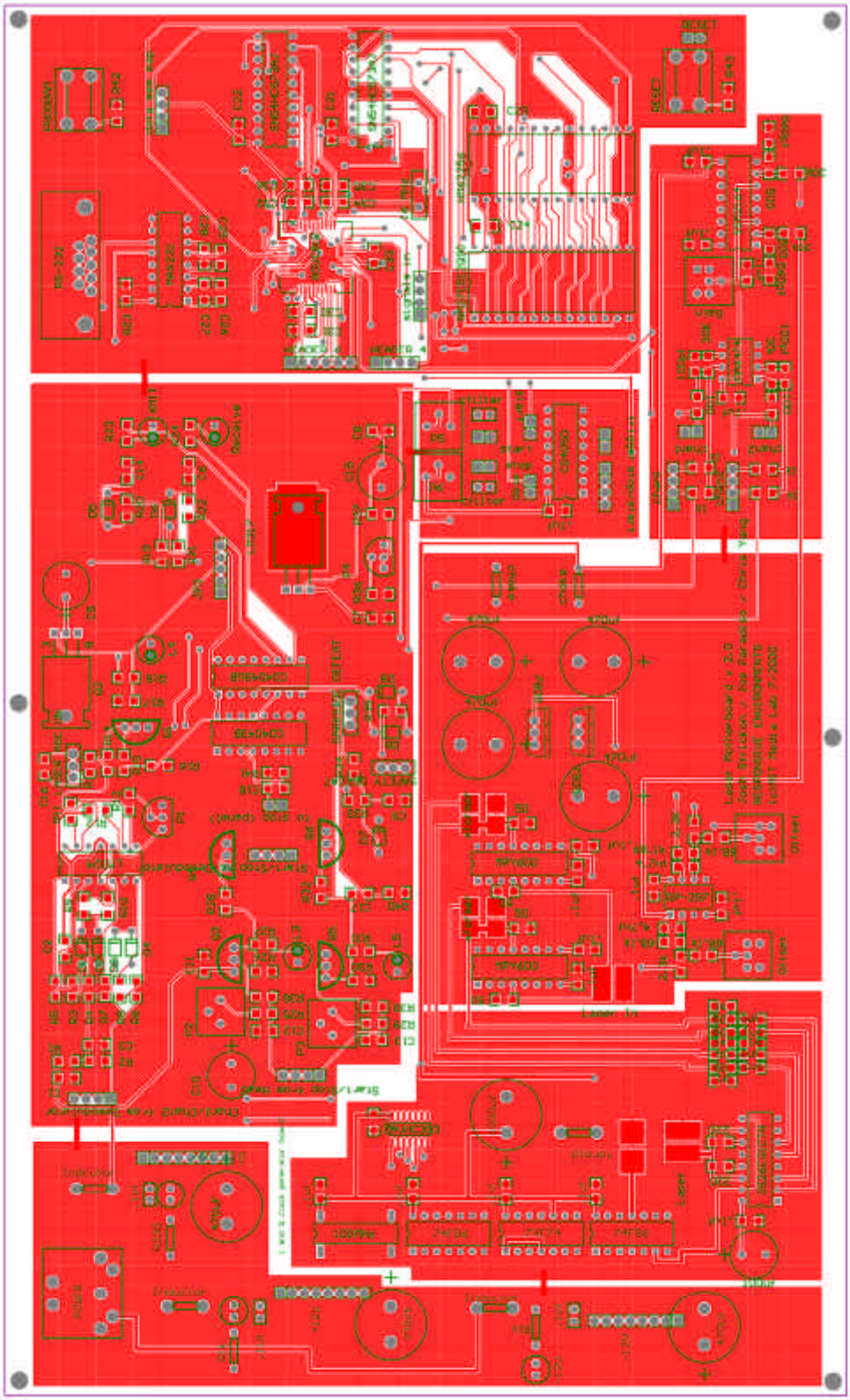
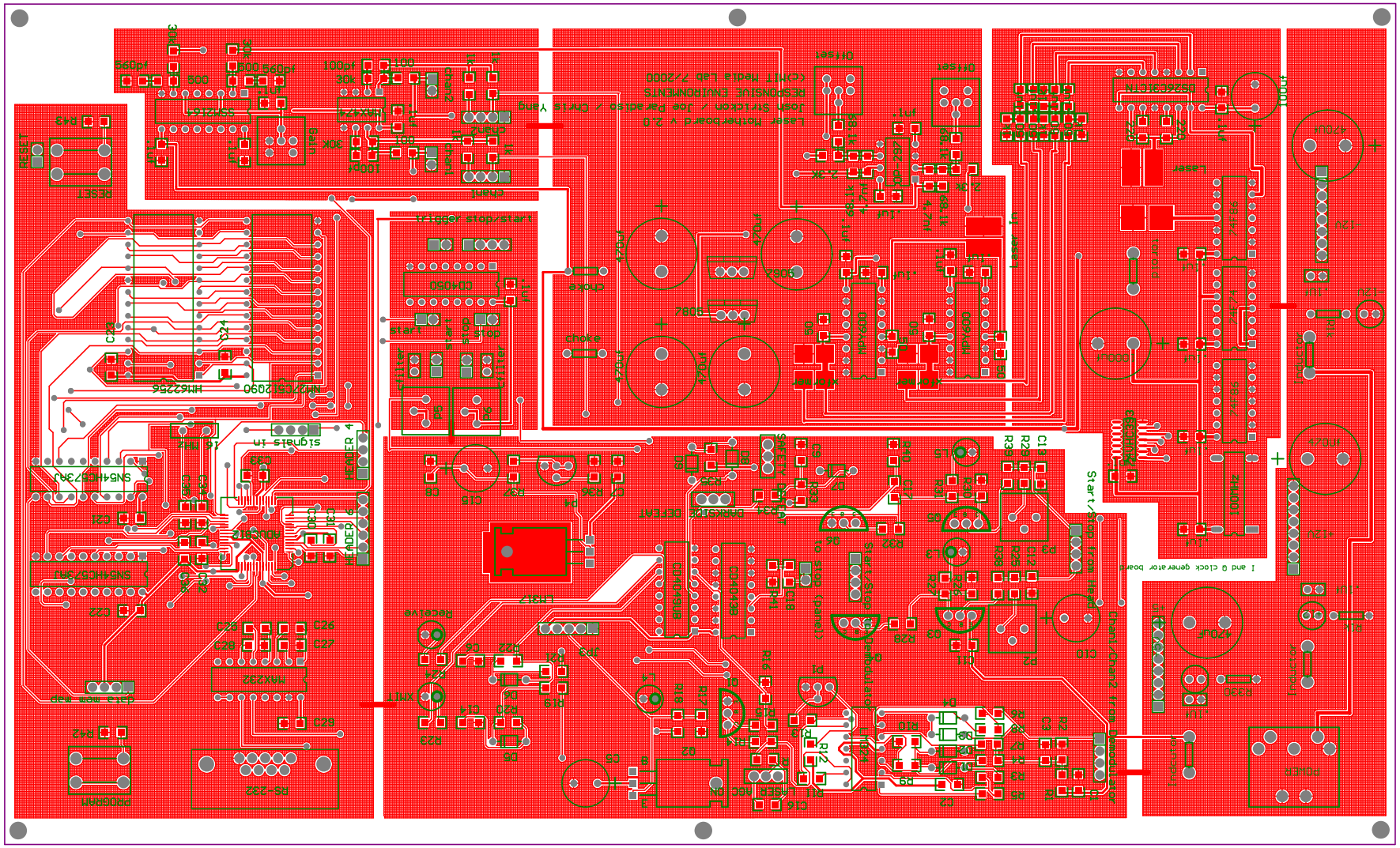


Figure A-4: Motherboard PCB: Top Layer



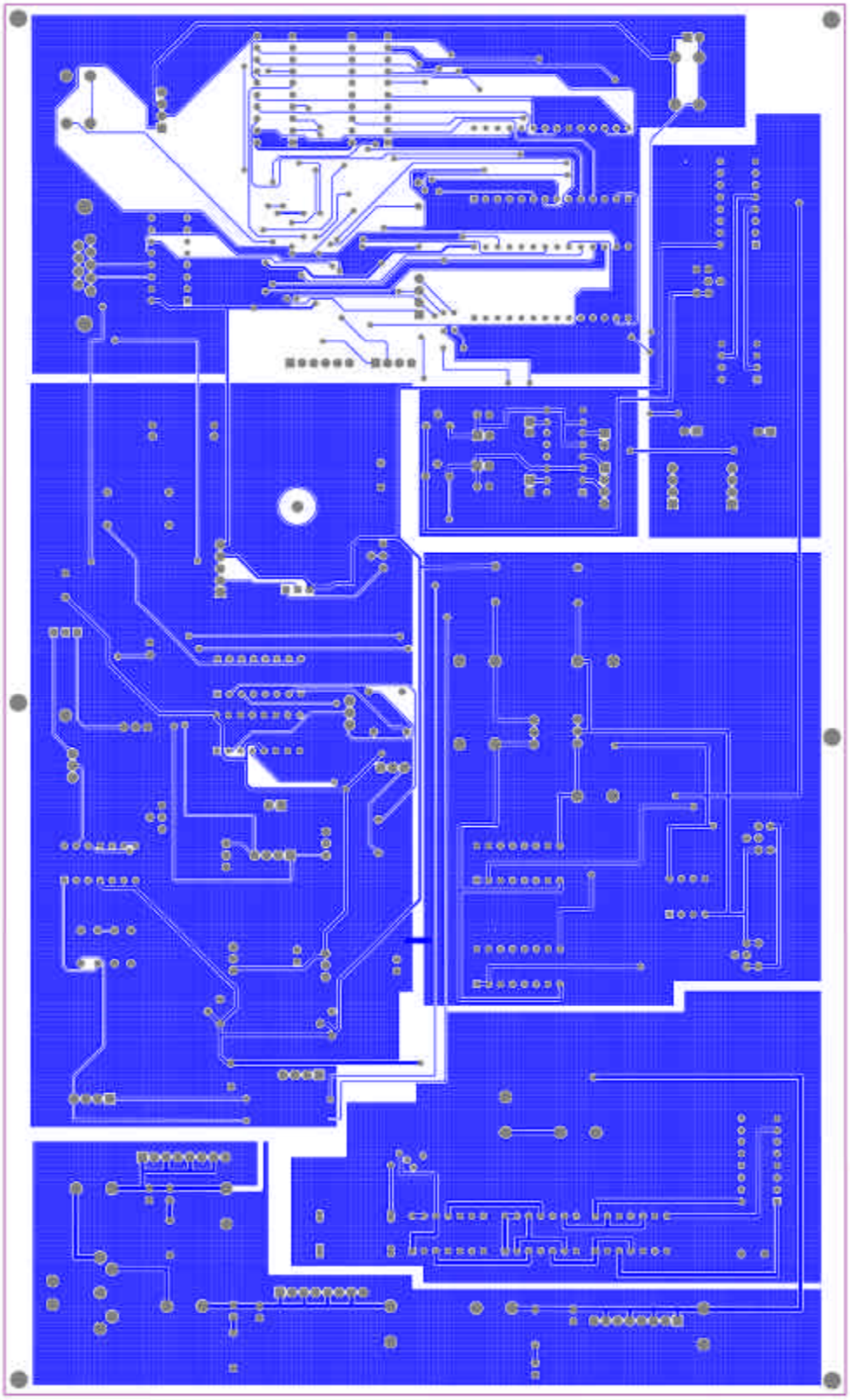
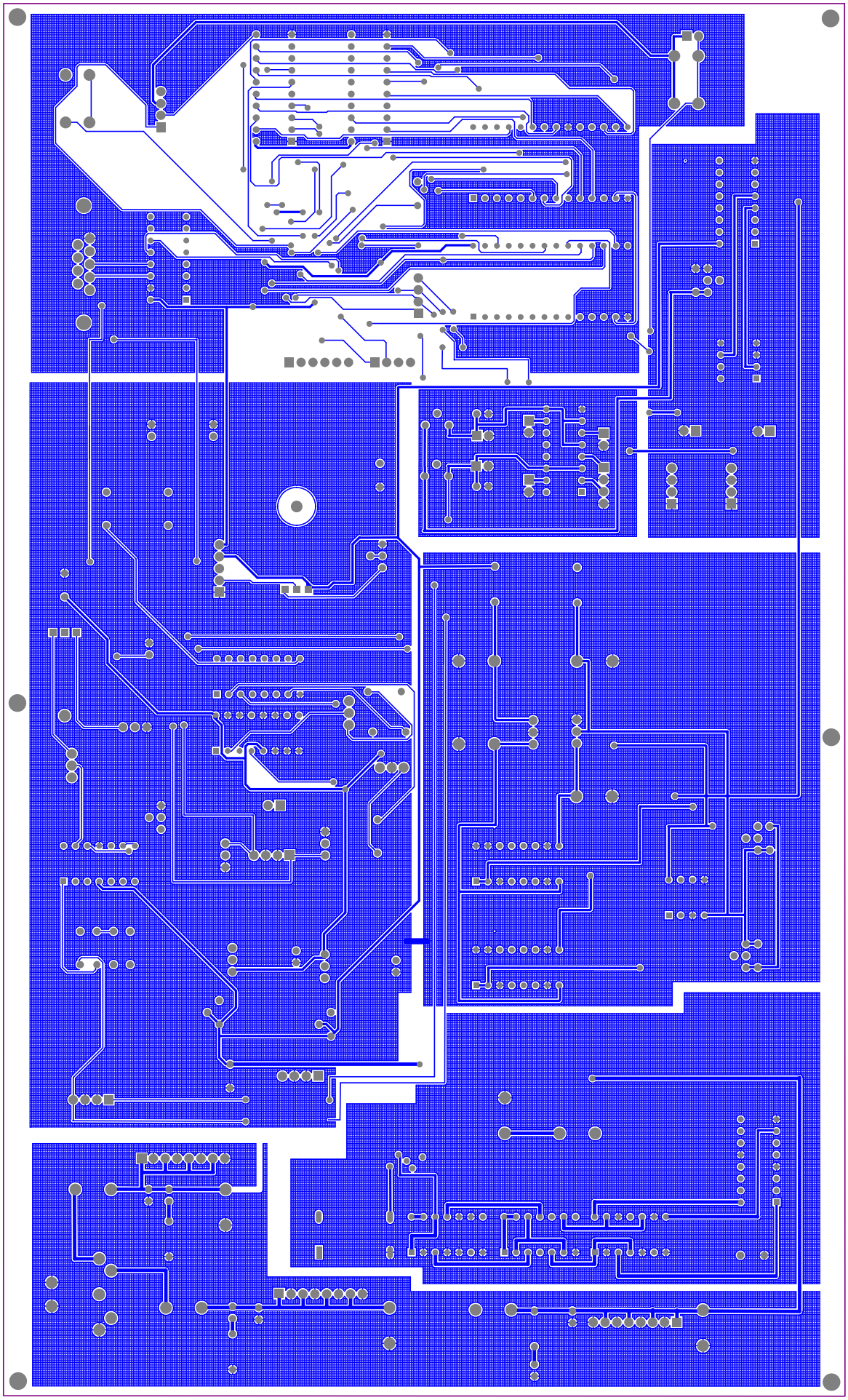


Figure A-5: Motherboard PCB: Bottom Layer



Appendix B

ADuC812 Embedded Code (C)

```
/* laser9.c - 14.400 kbps 1.14.01 */
/* optimized sampling for over 200 samples */
/* THIS IS THE TRULY CORRECT CODE - checked, double-checked, and checked again by yours
truly */
/* BUT 7.5Hz scan output rate (every fourth scanning cycle)  calc_scan is what takes up
the bulk of the time*/
/* last revised 1.14.01 */

#pragma DEBUG OBJECTEXTEND CODE
#include <ADuC812.h>
#include "812.h"
#include "laser9.h"
#include <stdlib.h>
#include <stdio.h>
#define BASESAMPLES 512

data bit debug=0;
int i_baseline[BASESAMPLES], q_baseline[BASESAMPLES];
int threshold_sum[BASESAMPLES];
unsigned int baseline_times[BASESAMPLES];
char base_samples[BASESAMPLES];
data int baseline_time;
int I_Samp[BASESAMPLES], Q_Samp[BASESAMPLES];
int Time_T[BASESAMPLES];

int kbhit() {
       if (RI==1)
              return 1;
       else return 0;
}

char getkey()  {
  char c;
  while (!RI);
  c = SBUF;
  RI = 0;
  return (c);
}


void init (void) {

// setup serial communication

//SCON = UART8BIT;
SCON = 0x50;
TMOD = T1TIMER | T18BITRELOAD | T0TIMER | T016BIT;
//SET_TIMER(1,0XFD, 0X00);      // 9600 baud
SET_TIMER(1,0XFA, 0X00);      // 7200 baud
PCON = DOUBLEBAUD;
TI=1;
START_TIMER(1);

printf ("UART initialized\n");

// set up ADC
printf ("setting up ADC\n");
ADCCON1 = ADCNORMAL | MCLKDIV4 | AQIN1;
printf ("ADCCON1 set\n");
ADCCON3 = 0x00;
printf ("ADCCON3 set\n");

EA=0;

}

void collectAnalog() // collect ADC output
{

       //int high;
       //int low;

       ADCCON2 = SINGLE | CHAN2;     // run ADC on chan1 (switched.. chan.1 connected to
ADC1)
       while (ADCCON3&0x80);
       STORE_ADC(I_Sample);          // collect chan1 ADC output

       ADCCON2 = SINGLE | CHAN1;     // run ADC on chan2 (ADC0)
       while (ADCCON3&0x80);
       STORE_ADC(Q_Sample);          //collect chan2 ADC output
}
```

```
void sample (void) { // sample I and Q and Timer 0
	collectAnalog();
	STORE_TIMER(0,Time_Temp);
}

void store_baseline(void) {  // find Baseline for i and q
	data int  i = 0, j = 0;
	int I_Base_Temp = 0;
	int Q_Base_Temp = 0;
	data int lastibase = 0, lastqbase = 0, lastbasetime = 0;
	I_Base = 0;
	Q_Base = 0;
	for (j=0; j<BASESAMPLES; j++) {
		base_samples[j] = 0;
		baseline_times[j] = 0;
		i_baseline[j] = 0;
		q_baseline[j] = 0;
		}
	while (STOP_SCAN){}
	while (!STOP_SCAN){}
	i=0;
	while (START_SCAN){
		sample();
		I_Base_Temp = I_Base_Temp + I_Sample.word;
		Q_Base_Temp = Q_Base_Temp + Q_Sample.word;
		++i;
	}

	I_Base = I_Base_Temp/i;
	Q_Base = Q_Base_Temp/i;

	for (j=0; j<8; j++) {
		i = 0;
		while (START_SCAN){}
		while (!START_SCAN){}
		SET_TIMER(0,0x00,0x00);
		while (STOP_SCAN){
			sample();
			I_Samp[i]=I_Sample.word;
			Q_Samp[i]=Q_Sample.word;
			Time_T[i]=Time_Temp.word;
			++i;
		}
		while(!STOP_SCAN) {
			sample();
			I_Samp[i]=I_Sample.word;
			Q_Samp[i]=Q_Sample.word;
			Time_T[i]=Time_Temp.word;
			++i;
		}
	}
	for (j=0; j<i; j++) {
		i_baseline[j] = i_baseline[j]+I_Samp[j];
		q_baseline[j] = q_baseline[j]+Q_Samp[j];
		baseline_times[j] +=(Time_T[j]>>3); // take average baseline (divide by 8)
		base_samples[j]++;
	}

	for (i=0;i<BASESAMPLES;i++) {
		if (base_samples[i]==0) {
			i_baseline[i]=lastibase;
			q_baseline[i]=lastqbase;
			baseline_times[i]=lastbasetime+baseline_times[i-1];
		}
		else {
			i_baseline[i]=i_baseline[i]/base_samples[i];
			q_baseline[i]=q_baseline[i]/base_samples[i];
			if (base_samples[i]!=8) {
				baseline_times[i]<<=3;
				baseline_times[i]/=base_samples[i];
			}
			lastibase=i_baseline[i];
			lastqbase=q_baseline[i];
			if(i==0)
				lastbasetime=0;
			else
				lastbasetime=baseline_times[i]-baseline_times[i-1];
		}
	}
//	for (i=0;i<BASESAMPLES;i++) {
```

```
//              printf("[%d %d %d]\n ", i, i_baseline[i], q_baseline[i]);
//              threshold_sum[i] = Threshold+i_baseline[i]+q_baseline[i];
//      }
}

void set_threshold(void) {
        int t;
        t=getkey();
        t=t<<8;
        Threshold=t+getkey();
        printf("Threshold set to %4X\r\n",Threshold);
}

void set_wait(void) {
        int t;
        t=getkey();
        t=t<<8;
        Wait_Time=t+getkey();
        printf("Wait Time set to %4X\r\n", Wait_Time);
}

void set_debug(void) {
        if (debug) {
                debug=0;
                printf("Debug Mode Off\r\n");
        }
        else {
                debug=1;
                printf("Debug Mode On\r\n");
        }
}

void output_data(void) {
        data short i=0;
        putchar(11);
        putchar(27);
        putchar(48);
        putchar(187);
        putchar(Num_Peaks);
        for (i=0;i<4;i++) {
                putchar(Ins[i].byte[0]);
                putchar(Ins[i].byte[1]);
                Ins[i].word=0;
                putchar(Quads[i].byte[0]);
                putchar(Quads[i].byte[1]);
                Quads[i].word=0;
                putchar(Time[i].byte[0]);
                putchar(Time[i].byte[1]);
                Time[i].word=0;
                putchar(Time_Start[i].byte[0]);
                putchar(Time_Start[i].byte[1]);
                Time_Start[i].word=0;
                putchar(Time_End[i].byte[0]);
                putchar(Time_End[i].byte[1]);
                Time_End[i].word=0;
                putchar(Ins_Sum[i].byte[0]);
                putchar(Ins_Sum[i].byte[1]);
                putchar(Ins_Sum[i].byte[2]);
                putchar(Ins_Sum[i].byte[3]);
                Ins_Sum[i].word=0;
                putchar(Quads_Sum[i].byte[0]);
                putchar(Quads_Sum[i].byte[1]);
                putchar(Quads_Sum[i].byte[2]);
                putchar(Quads_Sum[i].byte[3]);
                Quads_Sum[i].word=0;
        }
        Num_Samples=0;
        Num_Peaks=0;
}


void output_data_debug(void) {
        data short i=0;
        printf("Num_Peaks = %d\r\n",Num_Peaks);
        printf("Number of Samples = %d\r\n",Num_Samples);
        for (i=0;i<4;i++) {
                printf("i=%d ",i);
                printf("I=%d Q=%d ",Ins[i].word,Quads[i].word);
                printf("Time=%d.  Time_Start = %d. ",Time[i].word,Time_Start[i].word);
//              printf("Time_End=%d.  Time_Sample=%d.\r\n",Time_End[i],Time_Sample[i]);
                printf("I_Sum=%ld Q_Sum=%ld \r\n",Ins_Sum[i].word,Quads_Sum[i].word);
```

```
                Ins[i].word=0;
                Quads[i].word=0;
                Time[i].word=0;
                Time_Start[i].word=0;
                Time_End[i].word=0;
                Ins_Sum[i].word=0;
                Quads_Sum[i].word=0;
        }
        Num_Samples=0;
        Num_Peaks=0;
}

void calc_scan(void) {

        data int i=0;
//      P3 ^= 0x04;  // toggle pin P3.2
        for (i=0; i<Num_Samples; i++) {
        Time_Temp.word = Time_T[i];

        while (baseline_times[baseline_time]-Wait_Time<Time_Temp.word) {
                if (baseline_time>=BASESAMPLES-1) break;
                baseline_time++;
        }
        I_Temp=I_Samp[i]-i_baseline[baseline_time];
        Q_Temp=Q_Samp[i]-q_baseline[baseline_time];
        Sum_Temp=abs(I_Temp)+abs(Q_Temp);
//      Sum_Temp = I_Samp[i] + Q_Samp[i];

//      printf("\rS=%8X \n",Sum_Temp);
//      printf("Sum = %d \n",Sum_Temp);
//      printf("Threshold = %d \n",Threshold);

        if (Sum_Temp > Threshold) {
//              printf ("Sum_Temp>Threshold");
                //printf("I%d=%d-%d Q%d=%d-%d
t%d[%d].",I_Temp,I_Sample,i_baseline[baseline_time],Q_Temp,Q_Sample,q_baseline[baseline_t
ime];
                I_Sum_Temp.word =  I_Sum_Temp.word + I_Temp;
                Q_Sum_Temp.word =  Q_Sum_Temp.word + Q_Temp;



                // check if above threshold

                if (In_Peak)
                {
                        //check if currently in a peak
                        if (Sum_Temp>Sum_Max)
                        {
                                Sum_Max=Sum_Temp;
                                I_Max.word=I_Temp;            //if it is store it
                                Q_Max.word=Q_Temp;
                                Time_Max.word=Time_Temp.word;
                        }
                }
                else
                {
                        In_Peak=1;                      //now in peak set flag
                        Time_Start_Temp.word=Time_Temp.word;
                        Sum_Max=Sum_Temp;
                        I_Max.word=I_Temp;
                        Q_Max.word=Q_Temp;
                        Time_Max.word=Time_Temp.word;

                }
        }
        else
        {
                if (In_Peak)
                {               //out of peak for first time
//                      printf ("outta_Peak");
                        In_Peak=0;
                        Sum_Max=0;

                        //printf("num=%d\n",Num_Peaks);

                        if (Num_Peaks<4)
                        {
                                Ins[Num_Peaks]=I_Max;
                                Quads[Num_Peaks]=Q_Max;
                                Time[Num_Peaks]=Time_Max;
```

```
                                Time_Start[Num_Peaks]=Time_Start_Temp;
                                Time_End[Num_Peaks].word=Time_Temp.word;
                                Ins_Sum[Num_Peaks]=I_Sum_Temp;
                                Quads_Sum[Num_Peaks]=Q_Sum_Temp;
                                //Time_Sample[Num_Peaks]=tmp_time_sample;
                                ++Num_Peaks;
                                I_Sum_Temp.word=0;
                                Q_Sum_Temp.word=0;
                        }
                }
        }
    }
//      P3 ^= 0x04;
}

void read_data(void) {
        data int i=0;
        while(1) {
                while (START_SCAN){}
                while (!START_SCAN){}
                SET_TIMER(0,0x00,0x00);
                Time_Temp.word=0;
                while(Time_Temp.word<Wait_Time){STORE_TIMER(0,Time_Temp);}
                SET_TIMER(0,0x00,0x00);
                baseline_time=0;
                i=0;
                while(STOP_SCAN) {
//                      P3 ^= 0x04;
                        sample();
                        I_Samp[i]=I_Sample.word;
                        Q_Samp[i]=Q_Sample.word;
                        Time_T[i]=Time_Temp.word;
                        ++Num_Samples;
                        ++i;
                }
                while(!STOP_SCAN){
//                      P3 ^=0x04;
                        sample();
                        I_Samp[i]=I_Sample.word;
                        Q_Samp[i]=Q_Sample.word;
                        Time_T[i]=Time_Temp.word;
                        ++Num_Samples;
                        ++i;
                }
                calc_scan();
                while(kbhit())
                        if (getkey()=='A')
                                return;
                        In_Peak=0;

                        if(debug){
                                output_data_debug();
                        }
                        else {output_data();}
                        Sum_Max=0;
                        I_Sum_Temp.word=0;
                        Q_Sum_Temp.word=0;
        }
}

void wait_char(void) {
        unsigned int command;
        while (1) {
                if (kbhit()) {
                        while (kbhit()) {
                        command = getkey();
                        printf("g\n");
                        }
                        if (command=='A') {
                                printf("A\n");
                        }
                        if (command=='T') {
                                set_threshold();
                        }
                        if (command=='W') {
                                set_wait();
                        }
                        if (command=='D') {
                                set_debug();
                        }
                        if (command=='R') {
```

```
                                read_data();
                                debug=0;
                                return;
                        }
                }
        }
}

int main() {

        init();
        printf("initialized\n");
        printf("AD started\n");
        SET_TIMER(0,0x00,0x00);
        START_TIMER(0);
        printf ("storing baseline\n");
        while (1) {
                store_baseline();
                printf("baseline acquired\n");
                //printf("I_Base = %4X Q_Base = %4X\r\n",(I_Base,Q_Base));
                printf("I_Base = %d Q_Base = %d\n", I_Base, Q_Base);
                wait_char();
        }
        return 0;
}
```

```
// laser 9.h for laser9.c 01.14.01


data int i=0;
data int j=0;
data int Num_Samples=0;

lu Ins[4];
lu Quads[4];
du Ins_Sum[4];
du Quads_Sum[4];
lu Time[4];
lu Time_Start[4];
lu Time_End[4];
unsigned int Time_Sample[4];

data bit In_Peak=0;
data short Num_Peaks=0;
data int I_Temp;
data lu I_Sample;
data int Q_Temp;
data lu Q_Sample;

data lu Time_Temp;

data du I_Sum_Temp;
data du Q_Sum_Temp;

data lu I_Max;
data lu Q_Max;
data lu Time_Max;

data lu Time_Start_Temp;

data int Sum_Max=0;
data int Sum_Temp=0;
data int Threshold=0;
data unsigned int Wait_Time=0;

data int I_Base;
data int Q_Base;

#define START_SCAN     T0             //start phototransistor
#define STOP_SCAN      T1              //stop phototransistor
```

```
/* 812.h -- Stab at a collection of useful macros for the */
/*                                  ADuC812 microController.  */

/* Include SFR and sbit definitions */
#include "812SFR.h"

// Structures and unions
typedef union {
       int  word;
       unsigned char byte[2];
} lu;

typedef union {
       long word;
       unsigned char byte[4];
} du;

// RS232
/* Blocking send. Check to make sure that the previous send isn't still
   going, Reset TI bit, send. */
#define SEND(x)       while(!TI);TI=0;SBUF=x;
/* Non-blocking send. Check if previous send is done, if so, send */
//#define PUT(x)  if(TI) {TI=0;SBUF=x;}
#define PUT(x)


// Basic Timer Stuff
#define START_TIMER(num)      TR##num=1;
#define STOP_TIMER(num)              TR##num=0;
#define SET_TIMER(num,high,low)      TL##num=low;TH##num=high;
#define STORE_TIMER(num,un) un.byte[0]=TH##num;un.byte[1]=TL##num;
#define SET_TIMER2_RELOAD(high,low) RCAP2H=high;RCAP2L=low;


//ADC storage stuff
#define STORE_ADC(un)         un.byte[0]=ADCDATAH&0x0F;un.byte[1]=ADCDATAL;
//#define ADC_SETUP(void)             ADCCON1 = ADCNORMAL | MCLKDIV4; ADCCON3 = 0x00;
//#define ADC_SINGLE(num)             ADCCON2 = SINGLE | 0x0##num;

#define CONT 0x20
#define SINGLE 0x10
#define CHAN1 0x00
#define CHAN2 0x01
```

Appendix C

Two-Handed Application Code (C++)

```
// laser_interface.cpp

// this file defines the methods an application will use to talk to the
// laser range finder hardware through the laser_interface object

// C++ methods

// two-handed support for two objects put in, last updated 2.22.01 by chris yang


#include "laser_interface.h"
#include "serial2.h"
#include "test_main.h"


#define PI 3.14159265359
#define OFFSET 3 // define an offset to keep r from wrapping
#define SAMPLES 5
#define TWO_HANDED

#include <math.h>
#include <string.h>

double theta0[SAMPLES];
double theta1[SAMPLES];
float r0[SAMPLES];
float r1[SAMPLES];
float i0[SAMPLES];
float i1[SAMPLES];
float q0[SAMPLES];
float q1[SAMPLES];

double theta_threshold = 100;         // have to set
float r_threshold = 0.01;                     // have to set
double time_threshold = 300;// have to set
double start0;
double finish0;
double start1;
double finish1;
float r_velocity;
float i_velocity;
float q_velocity;
float min_r_velocity;         // have to set, coded out for now
double theta_velocity;
int blocked;
int countdown;
int myflag;
int zero_closer;
int time0;

laser_interface *li;

laser_interface::laser_interface(double in_x_range, double in_y_range)
{
        polling = true;
        user_x_range = in_x_range;
        user_y_range = in_y_range;
        comSetup();

        for(int i=0; i<MAX_SIGNALS; i++) {
                signals[i].I = 0;
                signals[i].Q = 0;
                signals[i].timep = 0;
                signals[i].timei = 0;
                signals[i].timef = 0;
                signals[i].i_sum = 0;
                signals[i].q_sum = 0;
                signals[i].theta = 0;
                signals[i].r = 0;
                signals[i].x = 0;
                signals[i].y = 0;
                signals[i].x_old = 0;
                signals[i].y_old = 0;
        }

        for(i=0; i<32; i++) {
                nl_coeff[i] = 0;
        }

        //load config file here
        FILE *fp;
```

```
        threshold = 0;
        wait_time = 0;
        fp = fopen("laser.cfg", "r");
        fscanf(fp, "%d %d", &threshold, &wait_time);
        set_threshold(threshold);
        //set_wait(wait_time);
        fclose(fp);
        sig_set_index = 0;
        sig_set_num = 0;
        collecting_data = false;
        load_nl_coeffs();
        rogu->tw.pr("got coeffs");

        blocked = 0; // clear blocked flag
        zero_closer = 1;

}

laser_interface::~laser_interface()
{
}

void laser_interface::msg(char *s)
{
       // insert your relevant printing message here
       rogu->tw.pr("%s", s);
}

void laser_interface::error_msg(char *s)
{
       // insert your relevant printing function here
       rogu->tw.pr("ERROR: %s\n", s);
}

void laser_interface::reset()
{
       polling = false;
       //char out_str[5];
       char in_str[200];
       char output[200];
       int in_len = strlen("Initialized SCI!\r\nCopied Data!\r\nLaser Online\r\nI_Base=
xxxx  Q_Base= xxxx\r\n");

       MessageBox(NULL, "Please reset the board now.", "LaserGL!", MB_ICONSTOP);

       read_bytes(in_len, (unsigned char *)in_str);
//     Communicate(NULL, 0, in_str, in_len, 0);

       // this scanf doesn't work right at all, but it doesn't matter
       // because we don't use these values anyway!
       int q, i;
       sscanf(in_str, "Laser Online\r\nI_Base= %4x Q_base= %4x\r\n", &i, &q);

       Ibase = i;
       Qbase = q;


       //sprintf(output, "RESET!   Qbase: %f Ibase: %f\n", Qbase, Ibase);
       sprintf(output, "%s\n\n", in_str);
       msg(output);

//     polling  = true;
}

void laser_interface::set_threshold(short int t)
{
       polling = false;
       char str[5];
       char reply[65535];
       char s[50];

       rogu->tw.pr("in set_thres");
       int len = strlen("g\ng\ng\nThreshold set to xxxx\r\n");

//     char tmp = t;

       eat_bytes();
       Sleep(1000);

       sprintf(str, "T%c%c", t/256, t%256);
```

```
        rogu->tw.pr("Threshold command:%d %d\n", t/256, t%256);

        write_data((unsigned char *)str);
        Sleep(1000);
        read_bytes(len, (unsigned char *)reply);

        sprintf(s, "thresh reply: %s\n", reply);
        msg(s);
rogu->tw.pr("done thres");
//      polling  = true;
}

void laser_interface::set_wait(short int w)
{
        polling = false;
        char str[5];
        char reply[40];
        char s[50];

        int len = strlen("Wait Time set to xxxx\r\n");

        sprintf(str, "W%c%c", w/256, w%256);
        rogu->tw.pr("wait command:%d %d\n", w/256, w%256);

        write_data((unsigned char *)str);

        read_bytes(len, (unsigned char *)reply);

        sprintf(s, "wait reply: %s\n", reply);
        msg(s);

//      polling  = true;
}


void laser_interface::readdata() {

//      msg("reading ");
        // first zero out the all of the signal structures
        for(int q=0; q<4; q++) {
                signals[q].I=0;
                signals[q].Q=0;
                signals[q].timep=0;
                signals[q].timei=0;
                signals[q].timef=0;
                signals[q].i_sum=0;
                signals[q].q_sum=0;
        }

        char str[100];

        read_data(str);

        char s[200];

        int i;
//      for (i=0; i<73; i++)
//              sprintf(s, "%d ", (int)str[i]);
//      sprintf(s, "\n");
//      msg(s);

        num_signals = str[0];
        if (num_signals>4) return;

        unsigned int tmp=0;
        unsigned int tmp1=0;
        unsigned int tmp2=0;
//      sprintf(s, "%d ", num_signals);
//              msg(s);

        for(i=0; i<num_signals; i++) {

                                // read in I as signed 16 bit and extend to 32
                                tmp = str[i*BLOCK];
                                if(tmp&0x80)    {signals[i].I    =    0xffff0000|    tmp<<8
|str[i*BLOCK+1+1];}
                                else{signals[i].I=tmp<<8 |str[i*BLOCK+1+1];}

                                // read in Q as signed 16 bit and extend to 32
                                tmp = str[i*BLOCK+2];
```

```
                                if(tmp&0x80){signals[i].Q     =     0xffff0000|     tmp<<8
|str[i*BLOCK+3+1];}
                                else{signals[i].Q=tmp<<8 |str[i*BLOCK+3+1];}



/*                              tmp=(str[i*BLOCK+4+1] <<8 );
                                signals[i].timep =  tmp | (str[i*BLOCK+5+1]& 0x000000ff);

                                tmp=(str[i*BLOCK+6+1] <<8);
                                signals[i].timei = tmp  | (str[i*BLOCK+7+1]&0x000000ff);

                                tmp=(str[i*BLOCK+8+1] << 8);
                                signals[i].timef =tmp | (str[i*BLOCK+9+1]&0x000000ff);
*/
        tmp=0x0000ffff & (str[i*BLOCK+4+1] << 8 );
  tmp1=0x0000ffff & (str[i*BLOCK+5+1]);
//  printf("%u %u\n", tmp, tmp1);
        signals[i].timep = tmp;//0x0000ffff & ( tmp | tmp1);
//      lasersignals[i].timep = 0x0000ffff & ( tmp | str[i*BLOCK+5+1]);

        tmp=0x0000ffff & (str[i*BLOCK+6+1] << 8);
        signals[i].timei = tmp;//0x0000ffff & ( tmp | str[i*BLOCK+7+1]);

        tmp=0x0000ffff & (str[i*BLOCK+8+1] << 8);
        signals[i].timef = tmp ;//0x0000ffff & ( tmp | str[i*BLOCK+9+1]);


                                tmp=(str[i*BLOCK+10+1] <<24);
                                tmp1=(str[i*BLOCK+11+1] <<16);
                                tmp2=(str[i*BLOCK+12+1] <<8);
                                signals[i].i_sum = tmp | tmp1| tmp2   | (str[i*BLOCK+13+1]&
0x000000ff);


                                tmp=(str[i*BLOCK+14+1] <<24);
                                tmp1=(str[i*BLOCK+15+1] <<16);
                                tmp2=(str[i*BLOCK+16+1] <<8);
                                signals[i].q_sum = tmp | tmp1| tmp2   |(str[i*BLOCK+17+1]&
0x000000ff) ;

                signals[i].theta = (double)((signals[i].timei + signals[i].timef)/2);
                signals[i].r = atan2(signals[i].i_sum, signals[i].q_sum);

                signals[i].r += OFFSET;

                if(signals[i].r>0){
                        signals[i].r=signals[i].r-2*3.14159;
                }

/*              if(signals[i].i_sum > 0 && signals[i].q_sum > 0) {
                        signals[i].good = false;
                }
                else
                {
                        */
                          signals[i].good = true;
                //}

        }
//for(i=0;i<2;i++){
//      sprintf(s, "Peaks=%d Numsignal=%d I:%d Q:%d i_sum:%d q_sum:%d r:%lf\n",
//                      num_signals,i,signals[i].I,     signals[i].Q,     signals[i].i_sum,
signals[i].q_sum,
//       signals[i].r);
//      msg(s); }

        // here we record the data for averaging and calibration
        if(collecting_data) {

        // first determine which is the biggest signal
                int current_biggest = 0;
                int biggest_index =0;

                //sprintf(s, "\nnum_signals: %d\n", num_signals);
                //msg(s);

                for(int i=0; i<num_signals; i++) {
                        if (signals[i].timei>63500) continue;
                        if(abs(signals[i].i_sum) + abs(signals[i].q_sum) > current_biggest)
{
```

```
                                current_biggest       =       abs(signals[i].i_sum)       +
abs(signals[i].q_sum);
                                biggest_index = i;
                        }
//              sprintf(s, "I:%d Q:%d  timep:%d  timei:%d timef:%d \ni_sum:%d q_sum:%d\n",
//                      signals[i].I, signals[i].Q, signals[i].timep, signals[i].timei,
//                      signals[i].timef, signals[i].i_sum, signals[i].q_sum);
//
//              msg(s);
                }

                if(num_signals>0 && sig_set_index < MAX_DATA_POINTS &&
                        signals[biggest_index].good) {

                        sig_set[sig_set_index].I = signals[biggest_index].I;
                        sig_set[sig_set_index].Q = signals[biggest_index].Q;
                        sig_set[sig_set_index].timep = signals[biggest_index].timep;
                        sig_set[sig_set_index].timei = signals[biggest_index].timei;
                        sig_set[sig_set_index].timef = signals[biggest_index].timef;
                        sig_set[sig_set_index].i_sum = signals[biggest_index].i_sum;
                        sig_set[sig_set_index].q_sum = signals[biggest_index].q_sum;
                        sig_set_index++;
                        if (rawfile!=NULL) {
                                fprintf(rawfile,    "%d    %d    %d    %d    %d    %d    %d\n",
signals[biggest_index].I,
                                        signals[biggest_index].Q,
                                        signals[biggest_index].timep,
                                        signals[biggest_index].timei,
                                        signals[biggest_index].timef,
                                        signals[biggest_index].i_sum,
                                        signals[biggest_index].q_sum);
                                fflush(rawfile);
                        }
                }
        }

}


void laser_interface::first_poll(){
        char s = 'R';

        write_bytes(1, (unsigned char*)&s);

}

// least squares stuff

void laser_interface::record_data_point(int n, float expected_r, float expected_t)
{
        char s[100];
        cal_data[n][0] = atan2(good_avg_i_sum, good_avg_q_sum);

        cal_data[n][0] += OFFSET; // to prevent wrap

        if(cal_data[n][0]>0){
                                cal_data[n][0]=cal_data[n][0]-2*3.14159;
                                }
        cal_data[n][1] = good_avg_t;

        sprintf(s,   "avg_i_sum:   %f   avg_q_sum:   %f   avg_t:   %f\n",   good_avg_i_sum,
good_avg_q_sum, good_avg_t);
        msg(s);
        sprintf(s, "translates to... r: %f theta: %f\n", cal_data[n][0], good_avg_t);
        msg(s);
        sprintf(s, "from %d samples\n", sig_set_index);
        msg(s);


        target_data[n][0] = expected_r;
        target_data[n][1] = expected_t;
}

void XYDivideMatrices(double A[NUMPOINTS][NUMSENSORS + 1],
                                        double B[NUMPOINTS],
                                        double X[NUMSENSORS + 1]);
void XYGaussJ(double a[NUMSENSORS +1][NUMSENSORS + 1]);

void laser_interface::calculate_xys()
{
        for(int i=0; i<num_signals; i++) {
```

```
                double  adj_r  =  signals[i].r*r_coeff[0]  +  signals[i].theta*r_coeff[1]  +
r_coeff[2];
                double        adj_theta        =        signals[i].r*theta_coeff[0]        +
signals[i].theta*theta_coeff[1] + theta_coeff[2];
                signals[i].x = adj_r*cos(adj_theta);
                signals[i].y = adj_r*sin(adj_theta);
        }
}

void laser_interface::quad_calculate_xys()
{
        for(int i=0; i<num_signals; i++) {
                double tr = signals[i].r*-1;
                double tt = signals[i].theta/2*(PI/180);
                double adj_r = q_r_coeff[0]*tr*tr + q_r_coeff[1]*tr + q_r_coeff[2]*tt +
                        q_r_coeff[3];
                double adj_theta = q_theta_coeff[0]*tr*tr + q_theta_coeff[1]*tr +
                        q_theta_coeff[2]*tt + q_theta_coeff[3];

                signals[i].x = adj_r*cos(adj_theta);
                signals[i].y = adj_r*sin(adj_theta);
        }
}

void laser_interface::nl_calculate_xys()
{
        double r, t, rsq, tsq, rcb, tcb;
        for (int i=0; i<num_signals; i++) {
                r = signals[i].r;
                t = signals[i].theta;

                rsq = r*r;
                rcb = rsq*r;
                tsq = t*t;
                tcb = t*t*t;
//   printf("%lf %lf\n", r, t);
                signals[i].x = nl_coeff[0] +
                        nl_coeff[1]*r +
                        nl_coeff[2]*t +
                        nl_coeff[3]*rsq +
                        nl_coeff[4]*rcb +
                        nl_coeff[5]*tsq +
                        nl_coeff[6]*tcb +
                        nl_coeff[7]*r*t +
                        nl_coeff[8]*rsq*t +
                        nl_coeff[9]*r*tsq +
                        nl_coeff[10]*r*tcb +
                        nl_coeff[11]*rcb*t +
                        nl_coeff[12]*rsq*tsq +
                        nl_coeff[13]*rcb*tsq +
                        nl_coeff[14]*rsq*tcb +
                        nl_coeff[15]*rcb*tcb;
                signals[i].y = nl_coeff[16] +
                        nl_coeff[17]*r +
                        nl_coeff[18]*t +
                        nl_coeff[19]*rsq +
                        nl_coeff[20]*rcb +
                        nl_coeff[21]*tsq +
                        nl_coeff[22]*tcb +
                        nl_coeff[23]*r*t +
                        nl_coeff[24]*rsq*t +
                        nl_coeff[25]*r*tsq +
                        nl_coeff[26]*r*tcb +
                        nl_coeff[27]*rcb*t +
                        nl_coeff[28]*rsq*tsq +
                        nl_coeff[29]*rcb*tsq +
                        nl_coeff[30]*rsq*tcb +
                        nl_coeff[31]*rcb*tcb;
        }
}

void laser_interface::calibrate_laser()
{
        double A[NUMPOINTS][NUMSENSORS + 1];
        int    i, j;
        double DesiredX[NUMPOINTS], DesiredY[NUMPOINTS];

        /* Set up the Matrix A: */
        for(i = 0; i < NUMPOINTS; i++) /* copy in the sensor data */
                for(j = 0; j < NUMSENSORS; j++)
                        A[i][j] = cal_data[i][j];
```

```
        for(i = 0; i < NUMPOINTS; i++) /* offset values for each point */
                A[i][NUMSENSORS] = 1.0;

        /* Set the desired values of X and Y
           The cal_data rows must match these rows
       in order for the points to match */
        SetDesiredValues(DesiredX, DesiredY);

        /* Calculate the x coefficients */
        XYDivideMatrices(A, DesiredX, r_coeff);

        /* Calculate the y coefficients*/
        XYDivideMatrices(A, DesiredY, theta_coeff);
}

void laser_interface::save_calibration()
{
        int i;
        char filename[25];
        char tmp[40];
        FILE* myfile;

        sprintf(filename, "laser.cal");

        myfile = fopen(filename,"w");
        if (myfile == NULL)
        {

                sprintf(tmp, "Unable to open %s. Calibration not saved!\n",filename);
                error_msg(tmp);
                return;
        }

        for(i=0;i<NUMXYCOEFFS;i++) // initialize sensor values
                fprintf(myfile, "%f\n", r_coeff[i]);
        for(i=0;i<NUMXYCOEFFS;i++)
                fprintf(myfile, "%f\n", theta_coeff[i]);

        fclose(myfile);
        sprintf(tmp, "Calibration saved to file %s\n",filename);
        msg(tmp);
}

void laser_interface::load_calibration()
{
        int i;
        char filename[25];
        char tmp[40];
        FILE* myfile;
        double d;

        sprintf(filename, "laser.cal");

        myfile = fopen(filename,"r");
        if (myfile == NULL)
        {
                sprintf(tmp, "Unable to open %s. Calibration not read!\n",filename);
                error_msg(tmp);
                return;
        }

        for(i=0;i<NUMXYCOEFFS;i++) // initialize sensor values
        {
                fscanf(myfile, "%lf", &d);
                r_coeff[i] = d;
        //      tw.pr("%f\n", d);
        }
        for(i=0;i<NUMXYCOEFFS;i++)
        {
                fscanf(myfile, "%lf", &d);
                theta_coeff[i] = d;
        //      tw.pr("%f\n", d);
        }

        fclose(myfile);

        sprintf(tmp, "Calibration loaded from file %s\n",filename);
        msg(tmp);
}

void laser_interface::load_quad_coeffs()
```

```
{
	int i;
	char filename[25];
	char tmp[40];
	FILE* myfile;
	double d;

	sprintf(filename, "laserq.cal");

	myfile = fopen(filename,"r");
	if (myfile == NULL)
	{
		sprintf(tmp, "Unable to open %s. Calibration not read!\n",filename);
		error_msg(tmp);
		return;
	}

	for(i=0;i<4;i++) // initialize sensor values
	{
		fscanf(myfile, "%lf", &d);
		q_r_coeff[i] = d;
	//	tw.pr("%f\n", d);
	}
	for(i=0;i<4;i++)
	{
		fscanf(myfile, "%lf", &d);
		q_theta_coeff[i] = d;
	//	tw.pr("%f\n", d);
	}

	fclose(myfile);

	sprintf(tmp, "Calibration loaded from file %s\n",filename);
	msg(tmp);
}

void laser_interface::load_nl_coeffs()
{
	int i;
	char filename[25];
	char tmp[40];
	FILE* myfile;
	double d;

	sprintf(filename, "pfit");

	myfile = fopen(filename,"r");
	if (myfile == NULL)
	{
		sprintf(tmp, "Unable to open %s. Calibration not read!\n",filename);
		error_msg(tmp);
		return;
	}

	for(i=0;i<32;i++) // initialize sensor values
	{
		fscanf(myfile, "%lf", &d);
		nl_coeff[i] = d;
		rogu->tw.pr("%f\n", d);
	}

	fclose(myfile);

	sprintf(tmp, "Calibration loaded from file %s\n",filename);
	msg(tmp);
}

void DoError(char *Error)
{
	char s[40];
	sprintf(s, "%s\n", Error);
	li->error_msg(s);
}

/* This sets the desired values Matrices for CalibrateGestureWall */
void	laser_interface::SetDesiredValues(double	DesiredR[NUMPOINTS],	double
DesiredT[NUMPOINTS])
{
	for(int i=0; i<NUMPOINTS; i++) {
		DesiredR[i] = target_data[i][0];
		DesiredT[i] = target_data[i][1];
```

```
        }


        //double tx, ty;

//      tx=0; ty = user_y_range;
//      DesiredR[0] = ty;  /* Upper left corner */
//      DesiredT[0] = tx;

//      tx=user_x_range/2; ty=user_y_range;
//      DesiredR[1] = sqrt(tx*tx+ty*ty); /* Upper middle */
//      DesiredT[1] = atan(tx/ty);

//      tx=user_x_range; ty=user_y_range;
//      DesiredR[2] = sqrt(tx*tx+ty*ty); /* Upper right corner */
//      DesiredT[2] = atan(tx/ty);
        /* ---------------------------------------- */
//      tx=0; ty=user_y_range/2;
//      DesiredR[3] = sqrt(tx*tx+ty*ty); /* Middle left side */
//      DesiredT[3] = atan(tx/ty);

//      tx=user_x_range/2; ty=user_y_range/2;
//      DesiredR[4] = sqrt(tx*tx+ty*ty); /* Middle */
//      DesiredT[4] = atan(tx/ty);

//      tx=user_x_range; ty=user_y_range/2;
//      DesiredR[5] = sqrt(tx*tx+ty*ty); /* Middle right side */
//      DesiredT[5] = atan(tx/ty);
        /* ---------------------------------------- */
//      tx=0; ty=0;
//      DesiredR[6] = sqrt(tx*tx+ty*ty); /* Lower left corner */
//      DesiredT[6] = PI/2;

//      tx=user_x_range/2; ty=0;
//      DesiredR[7] = sqrt(tx*tx+ty*ty); /* Lower middle */
//      DesiredT[7] = PI/2;

//      tx=user_x_range; ty=0;
//      DesiredR[8] = sqrt(tx*tx+ty*ty); /* Lower right */
//      DesiredT[8] = PI/2;
}

/* This divides matrices A and B and puts the result in X.
   This is used exclusively by CalibrateGestureWall
   Note that the Matrix dimensions are static. */
void XYDivideMatrices(double A[NUMPOINTS][NUMSENSORS + 1],
                                    double B[NUMPOINTS],
                                    double X[NUMSENSORS + 1])
{
        double ATranspose[NUMSENSORS + 1][NUMPOINTS];
        double ProductOfATransposeAndA[NUMSENSORS + 1][NUMSENSORS + 1];
        double ProductOfATransposeAndB[NUMSENSORS + 1];
        int    i, j, k;
        double Accumulator;

        /* Transpose the A matrix: */
        for(i = 0; i < NUMSENSORS + 1; i++)
                for(j = 0; j < NUMPOINTS; j++)
                        ATranspose[i][j] = A[j][i];

        /* Multiply ATranspose and A so we have a square matrix we can invert: */
        for(i = 0; i < NUMSENSORS + 1; i++)
        {
                for(j = 0; j < NUMSENSORS + 1; j++)
                {
                        Accumulator = 0; /* Reset the accumulator */
                        for(k = 0; k < NUMPOINTS; k++) /* take the dot product of the row
and column */
                                Accumulator += (ATranspose[j][k] * A[k][i]);
                        ProductOfATransposeAndA[j][i] = Accumulator;
                }
    }

        /* Invert the ProductOfATransposeAndA matrix: */
        XYGaussJ(ProductOfATransposeAndA);

        /* Multiply ATranspose with the B matrix of desired solutions */
        for(j = 0; j < NUMSENSORS + 1; j++)
    {
                Accumulator = 0; /* Reset the accumulator */
```

```
                for(k = 0; k < NUMPOINTS; k++) /* take the dot product of the row and
column */
                        Accumulator += (ATranspose[j][k] * B[k]);
                ProductOfATransposeAndB[j] = Accumulator;
    }

        /* Now we multiply ProductOfATransposeAndA with ProductOfATransposeAndB
           This is the final answer so we throw it in X */
        for(j = 0; j < NUMSENSORS + 1; j++)
    {
                Accumulator = 0; /* Reset the accumulator */
                for(k = 0; k < NUMSENSORS + 1; k++)
                /* take the dot product of the row and column */
                        Accumulator += (ProductOfATransposeAndA[j][k] *
                                        ProductOfATransposeAndB[k]);
                X[j] = Accumulator;
    }
}

#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;} /* Used by GaussJ */
/* Gets the Inverse of the square matrix A with NUMSENSORS + 1 rows
   Taken mostly from Numerical Recipies in C */
void XYGaussJ(double a[NUMSENSORS +1][NUMSENSORS + 1])
{
        int    *indxc, *indxr, *ipiv;
        int    i, icol = 0, irow = 0, j, k, m, mm;
        int    n = NUMSENSORS + 1;
        double big = 0, dum = 0, pivinv, temp;

        /* Allocate space for these guys */
        indxr = (int *)malloc(sizeof(int)*n);
        indxc = (int *)malloc(sizeof(int)*n);
        ipiv = (int *)malloc(sizeof(int)*n);

        for(j = 0; j < n; j++)
                ipiv[j] = 0;

        for(i = 0; i < n; i++)
    {
                big = 0;
                for(j = 0; j < n; j++)
                        if(ipiv[j] != 1)
                                for(k = 0; k < n; k++)
                                        if(ipiv[k] == 0)
                                        {
                                                if(fabs(a[j][k]) >= big)
                                                {
                                                        big = fabs(a[j][k]);
                                                        irow = j;
                                                        icol = k;
                                                }
                                        }
                                        else
                                        {
                                                if(ipiv[k] > 1)
                                                        DoError("Gaussj: Singular Matrix-1");
                                        }
                ++(ipiv[icol]);

                if(irow != icol)
                {
                        for(m = 0; m < n; m++)
                                SWAP(a[irow][m], a[icol][m]);
                }

                indxr[i] = irow;
                indxc[i] = icol;

                if(a[icol][icol] == 0)
                        DoError("GaussJ: Singual Matrix-2");

                pivinv = 1 / a[icol][icol];

                a[icol][icol] = 1;

                for(m = 0; m < n; m++)
                        a[icol][m] *= pivinv;

                for(mm = 0; mm < n; mm++)
                        if(mm != icol)
                        {
```

```
                                dum = a[mm][icol];
                                a[mm][icol] = 0;

                                for(m = 0; m < n; m++)
                                        a[mm][m] -= a[icol][m] * dum;
                        }
        }

                for(m = n-1; m >= 0; m--)
                {
                        if(indxr[m] != indxc[m])
                                for(k = 0; k < n; k++)
                                        SWAP(a[k][indxr[m]], a[k][indxc[m]]);
        }

                /* remember to free them!*/
                free(ipiv);
                free(indxr);
                free(indxc);
}

// signal averaging stuff

void laser_interface::begin_collecting_data()
{
        sig_set_index = 0;
        sig_set_num = 0;
        collecting_data = true;
}

void laser_interface::stop_collecting_data()
{
        collecting_data = false;
        dump_data_to_file();
        find_data_avg();
}


void laser_interface::dump_data_to_file()
{
        FILE *fp;
        fp = fopen("laser.dat", "w");

        for(int i=0; i<sig_set_index; i++) {
                fprintf(fp, "%f %il %il\n", .5*(sig_set[i].timef+sig_set[i].timei),
                        sig_set[i].i_sum, sig_set[i].q_sum);
        }

        fclose(fp);
}


void laser_interface::write_cal_data_to_file()
{
        FILE *fp;
        char s[200];
        fp = fopen("lasernew.dat", "w");
        for(int i=0; i<25; i++){
                fprintf(fp,   "%f   %f   %f   %f\n",   target_data[i][0],target_data[i][1],
cal_data[i][0], cal_data[i][1]);
        }
        fclose(fp);
        sprintf(s, "calibration data written to lasernew.dat\n");
        msg(s);

}

void laser_interface::find_data_avg()
{
        char s[40];
        sprintf(s, "Computing Avg...\n");
        msg(s);
        // calculate the first average
        float avg_q_sum = 0;
        float avg_i_sum = 0;
        float avg_time = 0;

        for(int i=0; i<sig_set_index; i++) {
                avg_q_sum += sig_set[i].q_sum;
                avg_i_sum += sig_set[i].i_sum;
                avg_time += .5*(sig_set[i].timef + sig_set[i].timei);
```

```
        }

        avg_q_sum = avg_q_sum/sig_set_index;
        avg_i_sum = avg_i_sum/sig_set_index;
        avg_time = avg_time/sig_set_index;

        // calcualte the variance
        float q_var = 0;
        float i_var = 0;
        float t_var = 0;

        for(i=0; i<sig_set_index; i++) {
                q_var += (sig_set[i].q_sum-avg_q_sum)*(sig_set[i].q_sum- avg_q_sum);
                i_var += (sig_set[i].i_sum-avg_i_sum)*(sig_set[i].i_sum-avg_i_sum);
                t_var += (.5*(sig_set[i].timef + sig_set[i].timei)-avg_time)*
                        (.5*(sig_set[i].timef + sig_set[i].timei)-avg_time);
        }

        q_var = q_var/sig_set_index;
        i_var = i_var/sig_set_index;
        t_var = t_var/sig_set_index;

        // select good points and calculate a new average
        float new_q_avg=0;
        float new_i_avg=0;
        float new_t_avg=0;
        int num_new_data=0;
        for(i=0; i<sig_set_index; i++) {
                if((sig_set[i].q_sum-avg_q_sum)*(sig_set[i].q_sum- avg_q_sum) <= q_var &&
                        (sig_set[i].i_sum-avg_i_sum)*(sig_set[i].i_sum-avg_i_sum)  <=  i_var
&&
                        (.5*(sig_set[i].timef + sig_set[i].timei)-avg_time)*
                        (.5*(sig_set[i].timef + sig_set[i].timei)-avg_time) <= t_var) {

                        new_q_avg += sig_set[i].q_sum;
                        new_i_avg += sig_set[i].i_sum;
                        new_t_avg += .5*(sig_set[i].timef + sig_set[i].timei);
                        num_new_data++;
                }
        }

        new_q_avg = new_q_avg/num_new_data;
        new_i_avg = new_i_avg/num_new_data;
        new_t_avg = new_t_avg/num_new_data;

        sprintf(s, "num_new_data: %i\n", num_new_data);
        msg(s);

        good_avg_q_sum = new_q_avg;
        good_avg_i_sum = new_i_avg;
        good_avg_t = new_t_avg;

}

void laser_interface::process_data(void){

        int i;

        float alpha = ALPHA;

        //The filter operating system generates values of xPos and yPos based on
        //what it thinks it was linked to the previous time.



        //peaks are same, just average
        if(num_signals == num_signals_old) {
                for(i=0;i<num_signals;i++) {

                        signals[i].x = alpha * signals[i].x_old + (1-alpha)*signals[i].x;

                        signals[i].y = alpha * signals[i].y_old + (1-alpha)*signals[i].y;
                }
        }
         //copy data over
        num_signals_old=num_signals;
        for(i=0; i<num_signals;i++){
                signals[i].x_old=signals[i].x;
                signals[i].y_old=signals[i].y;
        }
```

```
}

// interface functions


void initialize_rangefinder(double in_x_range, double in_y_range)
{
	li = new laser_interface(in_x_range, in_y_range);
	//li->reset();
}

void rangefinder_first_poll()
{
	li->first_poll();
}

void get_rangefinder_points(double *x, double *y, int *num_points)
{
	char s[200];

#ifdef TWO_HANDED
	update();
	if (li->num_signals == 2)
	{
		check();
	}
	extrapolate();
	update2();
#endif

	li->nl_calculate_xys();
	(*num_points) = li->num_signals;

	//li->process_data();
	for(int i=0; i<*num_points; i++) {

		x[i] = 640 - li->signals[i].x;
		y[i] = 480 - li->signals[i].y;
		sprintf(s,"i = %d r= %f theta=%f x=%f y=%f \n", i, li->signals[i].r, li-
>signals[i].theta, x[i], y[i]);
//		sprintf(s,"i = %d r= %f timei=%f timef=%f \n", i, li->signals[i].r,
(double)li->signals[i].timei, (double)li->signals[i].timef);
		li->msg(s);
	}

#ifdef TWO_HANDED
	fprintf(datafile, "blocked = %d i = %d r= %f timei=%f timef=%f \n", blocked, i,
li->signals[i].r, (double)li->signals[i].timei, (double)li->signals[i].timef);
	fflush(datafile);
#endif
	fprintf(pointsfile, "%f %f\n", x[0], y[0]);
	fflush(pointsfile);

}

void reset_rangefinder()
{
	li->reset();
}


void set_rangefinder_threshold(short int t)
{
	li->set_threshold(t);
}

void set_rangefinder_wait(short int w)
{
	li->set_wait(w);
}

void cleanup_rangefinder()
{
	delete li;
}

void write_data_file()
{
	li->write_cal_data_to_file();
}
```

```
void record_rangefinder_datapoint(int n, float x, float y)
{
//      li->record_data_point(n, sqrt(x*x+y*y), atan2(y, x));
        li->record_data_point(n, y, x);
}

void load_rangefinder_calibration()
{
        li->load_calibration();
}

void save_rangefinder_calibration()
{
        li->save_calibration();
}

void calibrate_rangefinder()
{
        li->calibrate_laser();
}

void begin_collecting_data()
{
        if(!(li->collecting_data)) {
        char s[40];
        sprintf(s, "beginning to collect data....\n");
        li->msg(s);
        li->begin_collecting_data();
//      li->polling=true;
        }
}

void stop_collecting_data()
{
        li->stop_collecting_data();
        char s[40];
        sprintf(s, "done collecting data!\n");
        li->msg(s);
//      li->polling=false;
}

void run()
{
        if(!(li->collecting_data)) {
        char s[40];
        sprintf(s, "running\n");
        li->msg(s);
        li->begin_collecting_data();
        li->polling=true;
        }
}

void rangefinder_readdata()
{
        li->readdata();
}

void update()
{
        int i;
        char s[200];
        for (i=0; i<(SAMPLES-1); i++)
        {
                theta0[i] = theta0[i+1];
                theta1[i] = theta1[i+1];
                r0[i] = r0[i+1];
                r1[i] = r1[i+1];
/*              i0[i] = i0 [i+1];
                i1[i] = i1 [i+1];
                q0[i] = q0 [i+1];
                q1[i] = q1 [i+1]; */
        }
        if (zero_closer==1)
        {
        theta0[SAMPLES-1] = li->signals[0].theta;
        theta1[SAMPLES-1] = li->signals[1].theta;
        r0[SAMPLES-1] = li->signals[0].r;
        r1[SAMPLES-1] = li->signals[1].r;
        i0[SAMPLES-1] = li->signals[0].i_sum;
        i1[SAMPLES-1] = li->signals[1].i_sum;
```

```
        q0[SAMPLES-1] = li->signals[0].q_sum;
        q1[SAMPLES-1] = li->signals[1].q_sum;
        start0 = (double)li->signals[0].timei;
        finish0 = (double)li->signals[0].timef;
        start1 = (double)li->signals[1].timei;
        finish1 = (double)li->signals[1].timef;
        }

        // if signal 1 is closer, then when signal 1 is blocking signal 0, signal 1 will
become signal 0 (since
        // the rangefinder won't see the second signal (originally signal 0).

        else
        {
        theta0[SAMPLES-1] = li->signals[1].theta;
        theta1[SAMPLES-1] = li->signals[0].theta;
        r0[SAMPLES-1] = li->signals[1].r;
        r1[SAMPLES-1] = li->signals[0].r;
        i0[SAMPLES-1] = li->signals[1].i_sum;
        i1[SAMPLES-1] = li->signals[0].i_sum;
        q0[SAMPLES-1] = li->signals[1].q_sum;
        q1[SAMPLES-1] = li->signals[0].q_sum;
        start0 = (double)li->signals[1].timei;
        finish0 = (double)li->signals[1].timef;
        start1 = (double)li->signals[0].timei;
        finish1 = (double)li->signals[0].timef;
        }
/*      for (i=0; i<SAMPLES; i++)
                {
                        sprintf(s, "theta0[%d]=%f r0[%d]=%f \n", i, theta0[i], i, r0[i]);
                        li->msg(s);
                }
        sprintf(s, "middle\n");
        li->msg(s);
        for (i=0; i<SAMPLES; i++)
                {
                        sprintf(s, "theta1[%d]=%f r1[%d]=%f \n", i, theta1[i], i, r1[i]);
                        li->msg(s);
                }
        sprintf(s, "end\n");
        li->msg(s);*/
}

// checks to see if one hand is blocking another signal, if so, set blocked flag=1
// if the end of one signal comes within a certain threshold (time) of another signals'
beginning, these signals
// will be seen as one, so assume blocked (if there were two signals to begin with)

void check()
{
        char s[100];
//      sprintf(s,  "finish0=%f  start1=%f  finish1=%f  start0=%f  \n",  finish0,  start1,
finish1, start0);
//      sprintf(s,  "sum1=%f  start1=%f  sum2=%f  start0=%f  \n",  finish0+time_threshold,
start1, finish1+time_threshold, start0);
        li->msg(s);
        if ((li->num_signals >= 2) && (blocked==0))
        {
               if (finish0 <= start1) // signal 0 earlier
               {
                       time0 = 1; // signal 0 earlier flag

                       if ((finish0 + time_threshold) >= start1)

                       // if end of one signal plus a certain time threshold is close to
the start of another signal

                       {
                               blocked=1;
                               myflag=1;
                               sprintf(s,"blocked \n");
                               li->msg(s);
                       }
               }
               else
               {
                       time0 = 0; // signal 1 earlier

                       if ((finish1 + time_threshold) >= start0)
                       {
                               blocked=1;
```

```
                                myflag=1;
                                sprintf(s,"blocked \n");
                                li->msg(s);
                        }
                }
        }
/*      else if (blocked == 1) {blocked = 1;}
        else
        {
                blocked = 0;
                countdown = SAMPLES;
        } */
} // not the way to check for blocked signals, because you have to remember it's blocked,
since li->num_signals will be 1.


// if there is only one signal AND that one signal is still in the same place (within
some threshold)
// AND the blocked flag is set, then assume there is a blocked signal
// if there is more than one signal, assume that the hand is no longer being blocked and
has moved elsewhere


void extrapolate()
{
        char s[200];

        if ((blocked==1) && (myflag==1))
        {
                if (r0[SAMPLES-1] < r1[SAMPLES-1]) {zero_closer=1;} // test to see if
signal zero is closer
                else {zero_closer=0;}
                myflag=0;
                countdown = SAMPLES;
                return;
        }
        // first time around, there will always be 2 signals,
        // and blocked will get reset as soon as it has been set. need some kind of memory
or flag. (myflag)

        else if ((blocked==1) && (li->num_signals >= 2))
        {
                blocked = 0;
                zero_closer=1;
//              countdown = SAMPLES;
        }

        /*if (blocked==1 &&
                (li->num_signals == 1) &&
                ((abs (theta0[0] - theta0[1])) <= theta_threshold) &&
                ((abs (r0[0] - r0[1])) <= r_threshold)) */

        else if ((blocked==1) && (li->num_signals == 1))  // for now just assume that the
blocking signal stays in one place
        {
                sprintf(s,"extrapolating\n");
                li->msg(s);

        // extrapolate
                li->num_signals = 2;
                //li->num_signals++;

                // for now, if countdown reaches zero, keep the signal there.
                if (countdown == 0)
                {
                        sprintf(s, "countdown = 0\n");
                        li->msg(s);
                        if (r0[SAMPLES-2] < r1[SAMPLES-2]) // check to see which one is
closer (blocking), closer one has more negative r
                        {
                                r1[SAMPLES-1] = r1[SAMPLES-2];
/*                              i1[SAMPLES-1] = i1[SAMPLES-2];
                                q1[SAMPLES-1] = q1[SAMPLES-2];*/
                                theta1[SAMPLES-1] = theta1[SAMPLES-2];
                        }
                        else
                        {
                                r0[SAMPLES-1] = r0[SAMPLES-2];
/*                              i0[SAMPLES-1] = i0[SAMPLES-2];
                                q0[SAMPLES-1] = q0[SAMPLES-2];*/
                                theta0[SAMPLES-1] = theta0[SAMPLES-2];
```

```
                        }

                        //li->num_signals = 1;
                }
                else
                {
                        if (r0[SAMPLES-2] < r1[SAMPLES-2])
                        {
                                sprintf(s, "signal 0 blocking signal 1, countdown = %d\n",
countdown);
                                li->msg(s);
                                r_velocity = ((r1[0] - r1[SAMPLES-2]) / (SAMPLES-1));
/*                              i_velocity = ((i1[0] - i1[SAMPLES-2]) / (SAMPLES-1));
                                q_velocity = ((q1[0] - q1[SAMPLES-2]) / (SAMPLES-1));*/
                                /*if (r_velocity < MIN_R_VELOCITY)
                                { r_velocity = MIN_R_VELOCITY; }*/
                                sprintf(s, "r_velocity = %f ", r_velocity);
                                li->msg(s);
                                r1[SAMPLES-1] = r1[SAMPLES-2] - r_velocity;
/*                              i1[SAMPLES-1] = i1[SAMPLES-2] + i_velocity;
                                q1[SAMPLES-1] = q1[SAMPLES-2] + q_velocity;*/
                                if (time0 == 1) // if signal 0 is earlier, then blocked
signal 1 is coming from left to right
                                {
                                        theta_velocity = ((start0-finish0) / (SAMPLES-1));
                                }
                                else
                                {
                                        theta_velocity = ((finish0-start0) / (SAMPLES-1));
                                }
                                sprintf(s, "theta_velocity=%f \n", theta_velocity);
                                li->msg(s);
                                theta1[SAMPLES-1] = (theta1[SAMPLES-2] + theta_velocity);
                                /*theta0[SAMPLES-1] = theta0[SAMPLES-2]; // to lock the
blocking signal so it doesnt gravitate towards blocked signal
                                r0[SAMPLES-1] = r0[SAMPLES-2];*/
                        }
                        else
                        {
                                sprintf(s, "signal 1 blocking signal 0, countdown = %d\n",
countdown);
                                li->msg(s);
                                r_velocity = ((r0[0] - r0[SAMPLES-2]) / (SAMPLES-1));
/*                              i_velocity = ((i0[0] - i0[SAMPLES-2]) / (SAMPLES-1));
                                q_velocity = ((q0[0] - q0[SAMPLES-2]) / (SAMPLES-1));*/
                                /*if (r_velocity < MIN_R_VELOCITY)
                                { r_velocity = MIN_R_VELOCITY; }*/
                                sprintf(s, "r_velocity = %f ", r_velocity);
                                li->msg(s);
                                r0[SAMPLES-1] = r0[SAMPLES-2] - r_velocity;
/*                              i0[SAMPLES-1] = i0[SAMPLES-2] + i_velocity;
                                q0[SAMPLES-1] = q0[SAMPLES-2] + q_velocity;*/
                                if (time0 == 1) // if signal 0 is earlier, then blocked
signal 0 is coming from right to left
                                {
                                        theta_velocity = ((finish1-start1) / (SAMPLES-1));
                                }
                                else
                                {
                                        theta_velocity = ((start1-finish1) / (SAMPLES-1));
                                }
                                sprintf(s, "theta_velocity=%f \n", theta_velocity);
                                li->msg(s);
                                theta0[SAMPLES-1] = (theta0[SAMPLES-2] + theta_velocity);
                                /*theta1[SAMPLES-1] = theta1[SAMPLES-2]; // to lock the
blocking signal so it doesnt gravitate towards blocked signal
                                r1[SAMPLES-1] = r1[SAMPLES-2];*/
                        }
                        countdown--;
                }
        }

        if (blocked==1)
        {
        sprintf(s,"blocked=1\n");
        li->msg(s);
        }
}

// after extrapolating, update real values.
```

```
void update2()
{
	int i;
	char s[200];
	li->signals[0].theta = theta0[SAMPLES-1];
	li->signals[1].theta = theta1[SAMPLES-1];
	li->signals[0].r = r0[SAMPLES-1];
	li->signals[1].r = r1[SAMPLES-1];

/*	for (i=0; i<SAMPLES; i++)
		{
			sprintf(s, "theta0[%d]=%f r0[%d]=%f \n", i, theta0[i], i, r0[i]);
			li->msg(s);
		}
	sprintf(s, "middle\n");
	li->msg(s);
	for (i=0; i<SAMPLES; i++)
		{
			sprintf(s, "theta1[%d]=%f r1[%d]=%f \n", i, theta1[i], i, r1[i]);
			li->msg(s);
		}
	sprintf(s, "end\n");
	li->msg(s);*/

/*	li->signals[0].r = atan2(i0[SAMPLES-1], q0[SAMPLES-1]);
	li->signals[1].r = atan2(i1[SAMPLES-1], q1[SAMPLES-1]);
	for (i=0; i<2; i++)
	{
	if(li->signals[i].r>0)
	{
			li->signals[i].r=li->signals[i].r-2*3.14159;
		}
	} */
//	signals[i].r = atan2(signals[i].i_sum, signals[i].q_sum);
}

// have to keep track of how many times extrapolated, you expect the hand to appear at
the other side by a certain time.
// if it doesn't, take the second object out.

// if countdown reaches zero... keep the blocked object in the same place until a new
signal appears.
```