MICROFLUIDICS PLATFORMS FOR QUANTITATIVE, MULTIPLEXED PROTEIN DETECTION

Thesis by

Habibullah Ahmad

In Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

CALIFORNIA INSTITUTE OF TECHNOLOGY

Pasadena, California

2011

(Defended February 23, 2011)

This work is dedicated to my mother,

Amina Ahmad

Acknowledgements

The pursuit of a PhD is one of the most ambitious goals I have ever undertaken. The past several years have been a period of intense scientific, intellectual, and personal growth for me; I have learned in equal measures new technical skillsets, how to conduct scientific research, and much about who I am as a person and who I want to be. The wonderful environment of Caltech has played a huge role in my growth, and I am deeply indebted to many people for this.

I would like to thank my advisor, Dr. Jim Heath, for the opportunity to work in his lab, and for working harder than any of his students to keep it well-funded. His focus on meaningful, immediate science afforded me great motivation throughout my studies, and his scientific vision is truly exceptional and inspirational. I would also like to thank Dr. Michael Ogawa, my undergraduate research advisor at Bowling Green State University. Dr. Ogawa was instrumental in kindling my interest in chemistry, and he provided me amazing opportunities while I worked in his lab. The impact of his nurturing and encouragement cannot be understated, and without him, I simply would not have pursued my studies at Caltech. I'm also indebted to Dr. Peter Willis and Dr. Kristen Beverly, who have remained the best of my mentors in the lab; they patiently guided me and helped me to get on my feet when I started here.

I've had the privilege of working with truly excellent colleagues throughout my studies. I'm particularly appreciative of Dr. Michael McAlpine, with whom I had a very productive working relationship that yielded several papers related to nanowire-based electronics on flexible substrates. Likewise, Dr. Rong Fan was also an immense credit to my scientific development and he worked with me extensively on DNA patterning projects. Together, Mike and Rong created an incredibly productive, synergistic, collaborative atmosphere which I hope to find or establish wherever I go. I would also like to thank Dr. Jun Wang, with whom I worked on a variation of the blood chip, Young Shik Shin, who worked with me on improving DNA flow patterning, and Alex Sutherland, who made concrete contributions to my microfluidic robotics project.

Scientific work aside, I would like to acknowledge all of my colleagues in the Heath lab over the years. With rare exception, they have made working here a genuine pleasure, and I count many of them among my personal friends. My office mates, Johnny Green, Akram Boukai, Rob Beckman, Yuri Bunimovich, John Nagarah, Ke Xu, Alex Sutherland, Himanshu Mishra, and Joey Varghese, deserve special mention. I'd like to thank Kevin Kan and Diane Robinson for all their cheerful help and friendship over the years; in addition, Dian Buchness, Agnes Tong, Steve Gould, and Joe Drew have all patiently attended to so many of my needs during my time here. I would like to specially thank Mike Roy, who was extremely helpful and meticulous with my machining needs at work, and was a personal friend outside the office.

I would also like to thank Mike Krout, who has been a close personal friend and (perhaps unknowingly) a continuous source of inspiration for me to persevere throughout my years here.

I must briefly make mention of some of the personal friends who have added color to my life during my time here. Rafed Al-Huq, Yusef Attia, Ali Gurel, Yernur Burketbeyevich Rysmagambetov, Saken Sherkhanov, Mansur Wadalawala, Mehmet Yenmez, and Bahattin Yildiz are among the best friends I have had, and they each supported me through the lows and helped me celebrate the highs.

Finally, and most importantly, I would like to thank Amina Ahmad and Amin Ahmad, my mother and my brother. They have both been unyielding sources of support and encouragement throughout my life, they have pushed me personally to achieve my full potential, and they have always looked out for my best interests. Both have made immense sacrifices in their own lives to give me the opportunity to pursue my PhD, and I am forever indebted and grateful to them.

Abstract

This thesis describes the development of microfluidic platforms that enable cheap, facile, rapid, and multi-parameter protein sensing. The first section of this work describes two strategies for high density DNA microarray patterning: microcontact printing and flow patterning. A protocol is provided for micron-scale alignment of multiple PDMS stamps to a single substrate, and a simple strategy to allow very low aspect-ratio stamping is enumerated.

The second section describes the formation of high density antibody microarrays using flow patterned DNA microarrays in conjunction with DEAL chemistry, and applies these microarrays to biological measurements. The platform's performance is first characterized using a human chorionic gonadotropin assay, and is subsequently used to stratify 22 cancer patients from frozen serum samples by quantifying the levels of twelve serum proteins. A microfluidic plasma separation device is then detailed to allow for similar measurements from fresh finger pricks of blood.

The third section of this work outlines improvements to the flow patterning platform through two alternate schemes: covalent attachment and DMSO patterning. Both protocols are shown to dramatically increase the consistency of microarray elements across a single chip when compared to the initial method. Theoretical simulations are used to describe the mechanism by which DMSO enhances patterning consistency.

The fourth section describes the design and fabrication of a robotics system that is capable of autonomously interfacing and manipulating PDMS substrates, and its application to producing barcode microarrays. The resulting substrates show unprecedented consistency from chip to chip, and we demonstrate through massively parallel single-cell measurements that data derived from different substrates is statistically indistinguishable.

Finally, we introduce an integrated software and hardware package designed to facilitate and automate microfluidic control at the laboratory level. We further provide the technical details of a related system which optimizes and comprehensively automates microfluidic blood assays such that even non-technical users who have never worked with microfluidics can regularly obtain the same standard of data that is produced in the lab.

Table of Contents

List of Tables and Figures

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 1

1.1 Introduction

Throughout the 1990s, the ongoing Human Genome Project promised to provide a quantum leap forward in our understanding of developmental and disease biology. While the genome did provide indispensable insight, it became clear in the decade that followed that the proteome was a far richer target in this regard. Although the genetic code may initially define a biological system, its subsequent contributions can vary wildly as a result of external factors that simply cannot be captured within the nucleobase sequence. Conversely, the proteome explicitly represents the end product of a system's configuration, and its characterization can yield a much greater understanding of what that system is trying to accomplish and how.

As the importance of measuring proteins came into focus in the past decade, so too did competing philosophies of how to study them. The previous half century was dominated by so-called "reductionist" biology, wherein scientists tried to understand the complexities of biological systems by breaking them down into their most basic subunits (i.e. proteins) and then exhaustively characterizing those individually[1]. Thus, pathway models described biological functions as discrete, autonomous collections of proteins that adhere to a rigid script of interactions to produce a desired outcome. However, these views are rapidly becoming outdated as it becomes increasingly clear

that proteins can have rich interaction profiles that entwine multiple, seemingly unrelated pathways into overarching protein networks. The central tenet of the systems biology philosophy, then, is that for any characterization of a protein to be meaningful, it must be placed in the context of its surrounding network[2]. Just as a DNA sequence alone cannot accurately predict the resulting system, nor can a single protein in isolation accurately describe a biological system's state.

This principle is readily illustrated in cancer systems. Broadly speaking, a cancerous state results when regulatory mechanisms of a cell become damaged by deletion, constitutive activation, etc. and the cell proliferates unchecked. However, the malfunction may be ascribed to damage in any of several pathways that converge upon that regulatory mechanism, and so a measurement of the latter alone will not necessarily provide actionable treatment information unless further enumerated[3]. Reciprocally, the systems biology approach predicts that a disruption in one part of a protein network will affect many other nodes at varying magnitudes. By simultaneously monitoring multiple nodes within a network, it may be possible to recognize "fingerprints" that are bespoke to a particular disease or variant thereof. Ideally, with regular monitoring, these perturbations may be detected before they grow to consequential levels. Indeed, positive treatment prognoses for many cancers are highly dependant on how early they are detected. Thus, embracing a systems biology approach to disease detection necessitates the development of cheap, multi-parameter

proteome diagnostics, and this is the driving principle for much of the work in this thesis.

Accurately characterizing protein expression levels is particularly challenging because, unlike DNA, proteins cannot be arbitrarily amplified. Among the known blood proteome, proteins can range in concentration from up to 10^9 pg/mL to as low as 1 pg/mL and below[4], and proteins found in the least abundance are suspected to be the most useful indicators in many cases. Cytokines, responsible for cell-to-cell communication, are almost exclusively in the 100 pg/mL and lower regime, as their local effective concentrations are highly diluted upon introduction to the main bloodstream. Detection of such rare proteins is further complicated by the overabundance of albumin, which can contribute to nonspecific fouling and constitutes a significant noise source for most types of measurements.

Today, the gold standard for quantitative protein measurements remains the ELISA. This assay uses a sandwich of antibodies to specifically immobilize and identify a target protein via a chromogenic readout. However, the technique scales very poorly and is ill-suited to the requirements of a systems biology-derived diagnostic on account of cost, sample consumption, and labor required. To fill this void, there have been concerted and very promising strides towards harnessing electronic measurements to detect proteins via chemically gated field effect transistors (FETs)[5], impedance spectroscopy[6], and giant magnetoresistance techniques[7], among others. Indeed, much of my early

work was aimed at producing functionalized silicon nanowire sensors, although those results are not included here. Instead, we focused on traditional optical detection and developed an assay that leverages the ELISA concept, but adds small refinements that allow the creation of high density antibody microarrays which spatially distinguish individual protein assays. Specifically, the DNA Encoded Antibody Library (DEAL) technique decorates capture antibodies with unique sequences of ssDNA; when introduced to a standard DNA microarray, a mixture of such antibodies self-assemble via complementary hybridization so that each DNA spot assays a unique protein[8]. Following analyte capture, the assay is completed with biotinylated secondary antibodies, which are then developed with fluorescently-tagged streptavidin. A standard microarray scanner quantitates the fluorescent readout.

The development of DEAL provided an opportunity to simultaneously perform multiplexed protein sandwich assays at drastically higher densities than the 96-well plates utilized for traditional ELISAs. Using a standard DNA microarray as a substrate meant that each protein assay was performed within a 150µm spot, and was separated by just 150µm from the next assay. When coupled with microfluidic technology, this newfound density enabled massively parallel protein measurements from rare samples. Yet, on the scales of either microfluidics or biology, a 300µm pitch can hardly be considered “dense”; microchannels are readily fabricated in the low-micron range, while cells typically range from one to five microns in size. Thus, we quickly initiated an effort

to generate DNA microarrays that matched these dimensions in the hopes of enabling novel experiments at the single cell level.

1.2 DNA Patterning

In order to serve as a practical replacement for the larger, traditionally-spotted DNA microarrays, our new substrates had to fulfill several requirements. Fundamentally, it was critical to have arbitrary control over feature size and morphology in the low micron range. Ideally, the new patterning method should also not be constrained by a rigid array architecture, i.e. it should allow for irregular spacing among features, and it should also have some provision for positional control of patterned elements. These latter requirements are important if elements of the array need to interact with predefined features on a substrate or within a microfluidic circuit. Among the practical constraints was that the patterning procedure be relatively rapid, easy to execute, and ideally it could be performed in our own lab. Finally, a low-cost solution was preferred.

Microcontact printing (μCP), one of a battery of soft lithography techniques developed by the Whitesides group[9,10], emerged as an ideal candidate which fulfilled most of the aforementioned requirements. The process is directly analogous to a macroscale rubber stamp, wherein ink is applied to a featured surface which is then brought into contact with a substrate; the ink is only transferred along the raised features of the

stamp (Figure 1.7.1a). By substituting a PDMS device for the rubber stamp, μCP inherits all the dimensional and morphological flexibility of long-standing microfabrication techniques. Moreover, because the technique is parallel in nature, patterning thousands of feature instances for an array takes no more time than patterning a single one. Finally, μCP fares well from a cost perspective.

The obvious potential of this technique in the DNA microarray arena led to a 2004 report of high quality, patterned DNA deposition with feature sizes as small as 1μm via μCP. However, the technique involved a cumbersome, 45-minute inking process which reduced its viability for high throughput production of multi-component microarrays[11]. This was closely followed by a report detailing a much faster and more convenient procedure wherein DNA first adheres to the hydrophobic PDMS stamp via van der Waals interactions with its bases, and is then efficiently transferred to a positively-charged substrate via electrostatic interactions along the phosphate backbone[12]. This latter work inspired our efforts, and we quickly reproduced its results within the lab (Figure 1.7.1b). However, two unresolved technical issues prevented the platform's immediate adoption for DEAL experiments: stamp alignment and low aspect ratio feature printing.

1.3 Technical Issues

Creation of a multi-component DNA library via μCP is predicated upon the ability to align multiple stamps precisely to their target substrate (Figure 1.7.2a); as the feature density shrinks, so too does positioning tolerance. While solutions for this requirement have recently been reported[13-15], at the time it remained an open problem. We tackled the issue by capitalizing on the fine alignment capabilities of a Karl Süss MA-6 photolithography apparatus, which is designed to facilitate micron-scale alignment between photomasks and substrates. Ideally, an inked PDMS stamp would simply replace or be affixed to the photomask, while our microarray substrate would be placed on the wafer chuck below. However, when initiating alignment, the MA-6 performs a mandatory "wedge error correction" (WEC) routine wherein the substrate is briefly brought into contact with the photomask to ensure that the two are parallel before it drops down to the specified alignment gap. Thus, the substrate would be inked without any opportunity for alignment.

To circumvent this problem, the photomask was replaced by a precisely machined 1/8" thick plate bearing a central cutout. At the same time, the PDMS stamps were cast using a special aluminum stencil which creates a two-tiered substrate: the bottom tier's dimensions correspond to the photomask plate's cutout, but it is marginally (ca. 25μm) thicker and bears the desired microfeatures on its underside; the top tier simply provides a broad lip that is used as a handle. Figure 1.7.2b depicts the molding process

and the resulting stamp structure. In practice, substrates are loaded into the MA-6 and allowed to perform WEC against the bare photomask plate. Once the alignment gap is established, an inked stamp is inserted into the plate's cutout; the stamp's lip precisely positions the lower tier's microfeatures slightly below the plate surface. At this stage, fine alignment can be achieved by matching corresponding fiducials on the optically-transparent stamp and the substrate underneath via the MA-6's micromanipulators. Printing is accomplished by slowly reducing the alignment gap until the substrate and stamp make contact, as readily evidenced by a contrast change in the stamp features. Finally, the alignment gap is re-introduced, the spent stamp is lifted out, and the system is ready to load the next inked stamp. Once WEC is performed, the entire loading and alignment procedure for subsequent stamps generally requires only a couple of minutes.

The MA-6-based approach to multiple stamp alignment proved a satisfactory solution, exhibiting low-micron alignment precision and a fast, cheap, and non-demanding protocol. Figure 1.7.3 demonstrates the quality of alignment achieved across a variety of μCP patterns. Indeed, the solution proved so robust that minor variations of it have subsequently been employed to align densely-featured fluidic control and flow layers during PDMS fabrication, to align completed microfluidic molds with finely-featured Silicon substrates[16], and even to position SNAP[17] nanowire masters onto their target wafers.

The second major technical challenge related to microcontact printing of DNA microarrays derives from the deformable nature of PDMS. Specifically, feature height becomes an important parameter that must be carefully tuned according to feature size and feature density to prevent aberrant ink transfer[18,19]. Excessively tall features (high aspect ratio) are prone to tearing upon demolding, and can buckle or collapse laterally during stamping. Conversely, shorter features (low aspect ratio) are resistant to those failure modes but become prone to "roof collapse," wherein recessed areas between features sag or collapse onto the substrate (Figure 1.7.4). Most approaches to mitigating these issues focus on low aspect ratio features and make additional provisions to prevent roof collapse. The most basic such strategy is to simply add broad support structures in close proximity to small features of interest[11]. However, this still results in extraneous, though controlled, ink transfer, and is clearly not an ideal solution when creating large, high-density microarrays. An alternative solution is to utilize customized, harder formulations of PDMS[20-22] that are more resistant to deformation; these result in significantly better feature fidelity which extends well into the sub-micron range.

We developed a method for stamping low aspect ratio features that avoids specialty materials and is trivial to integrate into the standard PDMS fabrication workflow. By introducing a rigid material within the body of the stamp, the degree of deformation allowed at the stamp surface is significantly reduced. We implemented this solution by dicing a standard glass slide for use as the rigid support; after pouring PDMS prepolymer

into our aluminum casting stencil (Figure 1.7.3a) and degassing, the glass support is introduced parallel to the underlying wafer and pushed firmly to the bottom of the stencil. The result is an exceedingly thin layer of PDMS along the bottom of the stamp which is chemically adhered to the rigid glass slide during the curing process, leaving little room for unwanted deformation or sagging. We found this to be an excellent solution in the low micron regime relevant to our microarray fabrication, but did not perform limit testing to determine if the benefits extend to sub-micron features. For our purposes, the reinforced stamps were easily able to pattern 5µm-tall features at 1mm intervals – a lateral aspect ratio of 200:1 – without any threat of roof collapse. This represents a significant advance over unmodified stamping limits, and eliminates the last technical hurdle for practical microarray production via microcontact printing.

1.4 Chemistry

With our mechanical limitations resolved, we began generating microarrays tailored to investigate single-cell secretions. The goal was to create a large array of "bulls-eye" structures wherein the central spot of each would, using DEAL reagents, immobilize a single cell while the surrounding rings captured its secreted cytokines (Figure 1.7.4). However, we quickly found that stamped microarrays behaved very differently than spotted ones when used for DEAL assays. An investigation using fluorescent reagents demonstrated that the capture antibodies were not assembling as intended; indeed

they formed a completely inverted pattern wherein the complimentary DNA spot was not populated at all while the surrounding background areas were intensely patterned (Figure 1.7.5a).

After considerable study, we found the behavior was an indirect result of contaminants that leach from our PDMS stamps and are co-deposited with DNA; during the initial blocking step of the DEAL process, BSA is preferentially recruited to these contaminants and very efficiently prevents subsequent assembly of the capture antibody. The finding was not unprecedented[23,24], particularly among polar inks[25], and we tried a slew of methods to suppress it. A lengthy swelling procedure[26] designed to remove un-crosslinked monomers from bulk PDMS failed to alleviate the problem. Attempts to mask the contaminants by adding a fluoropolymer coating (DuPont Teflon AF) or patterned photoresists to the stamp surface yielded a sharp decrease in feature fidelity and degraded many of the PDMS's physical characteristics required for μCP.

Rather than remove the PDMS contaminants, an alternative strategy lay in omitting BSA from our DEAL protocol. While other biological blocking agents, such as casein, yielded similarly inverted patterns, we found that PEGylating the substrate did not inhibit capture antibodies from hybridizing with their target DNA spots. However, PEG also proved insufficient for blocking non-patterned areas effectively, as electrostatic interactions between the capture antibody's DNA and the aminated surface yielded significant non-specific binding. This was eliminated by back-filling the PEGylated

substrate with acetic anhydride, yielding a negatively-charged carboxylate surface. The combination of surface treatments finally provided a μCP-generated DNA microarray that was usable for DEAL experiments (Figure 1.7.5c), but due to lingering unease about PDMS contamination, concerns about DNA loading, and the concurrent development of an alternative patterning technique, we did not push this technology forwards.

1.5 Flow Patterning

The strikingly inverted images that we first obtained when performing DEAL assays on stamped microarrays inspired a new approach to DNA patterning. If regions of our substrate which had come into contact with PDMS and BSA were particularly resistant to further protein aggregation, while non-contacted areas readily adsorbed DNA-laden antibodies despite BSA blocking, why not invert the paradigm? Here, a PDMS slab would contact all the "background" areas of the substrate while maintaining recessed regions that correspond to the desired microarray features. Put simply, a PDMS device bearing channels would be bonded to a substrate and the channels filled with DNA solutions, thereby depositing DNA according to the channel morphology. Thus, in a somewhat convoluted way, the very simple idea of flow patterning was conceived.

Initial attempts at flow patterning relied on electrostatic interactions with a positively-charged substrate to immobilize DNA; shortly after filling each channel, the DNA

solutions were flushed away and rinsed with PBS buffer. While fluorescently-tagged oligomers indicated that the technique produced the expected patterns, DEAL experiments revealed that an insufficient amount of DNA was immobilized via this procedure, as evidenced by poor assay sensitivities. Consequently, we allowed our patterning solutions to evaporate and thereby deposit a significant fraction of their DNA on the substrate surface. This was followed by thermal or UV[27] crosslinking and produced densely-loaded patterns as desired. Moreover, the contamination principle from our μCP experiments held, and the flow-patterned arrays exhibited extremely low background during assays.

The flow patterning method is subject to a unique set of advantages and disadvantages when compared with μCP. Chief among the former is certainly the useful contaminant distribution, but there are additional benefits as well: because the procedure is an evaporative one, the amount of DNA deposited can be directly tuned by altering the patterning solution's concentration – a relationship which was much more tenuous with μCP. In addition, alignment issues amongst microarray elements become moot, as they are all defined monolithically with photolithographic precision. However, flow patterning is hamstrung by the fact that its reagent channels must be topologically continuous; when implemented in traditional 2D microfluidics, this prohibits discontinuous features such as traditional microarray spots and severely limits the scope of potential microarray architectures.

We sought to address this limitation by developing a 3D microchannel network, and targeted production of the same bulls-eye structure patterned earlier via μCP. The key component required for such a network is the crossover channel which transfers fluid between the upper and lower layers of a 2-layer network. Although multiple methods for accomplishing this have been reported [28-30], we developed a very simple protocol which does not require any extra steps during PDMS fabrication. Specifically, we generated a two-level lower flow layer from SU-8 wherein flow channels were patterned at 25μm height and crossover points were patterned as 50μm-tall posts that overlapped them. The upper flow layer was simply patterned at a uniform 25μm height. During device fabrication, the lower layer was spin-coated with PDMS at 5000 RPM for 60 seconds, yielding a very thin layer of PDMS. It is unclear if the tall posts protrude from the thin layer at this stage, or if they are covered by a thin membrane which is ruptured during curing (as the PDMS shrinks) or demolding. In any case, standard 2-layer protocols applied to these photoresist masters yield functional interlayer vias in unit yield. Figure 1.7.6 demonstrates successful implementation of a 3D flow patterning network to create a microarray comprised of 49 discontinuous, 3-element bulls-eye features.

The bulls-eye patterns, though a strong technical demonstration of our patterning capabilities, were never utilized for single-cell secretion studies. Among its demerits, it proved to be an inefficient architecture which would not scale well as more array elements were added. Instead, a very simple 2D flow patterning design, known as the

"barcode" microarray, gained traction as a high density, trivially fabricated alternative. These barcode substrates have unlocked a unique opportunity in our lab to measure multiple intracellular and secreted proteins from single cells, and they have played a fundamental role in several publications which are not detailed in this thesis. They continue to constitute an instrumental component in the majority of our lab's ongoing biological projects.

1.6 Thesis Overview

This thesis discusses the development of the flow patterned DNA microarrays into a robust platform that is capable of supporting accurate, consistent, and convenient bioassays for clinical diagnostics. Chapter 2 introduces the aforementioned barcode morphology and demonstrates its utility as a substrate for the DEAL platform. We demonstrate that the sensitivity of DEAL assays is directly dependant on the substrate's DNA loading, and take advantage of this fact to measure proteins with a dynamic range of over five orders of magnitude. The technology is validated by correctly determining the levels of human chorionic gonadotropin (hCG), a common pregnancy marker, from two serum samples in a blind test. We then apply this platform to the analysis of a dozen proteins from frozen cancer-derived serum samples. Finally, we adopt a microfluidic circuit for blood plasma separation and use it in conjunction with our barcodes to measure multiple proteins from small volumes of fresh, finger prick-derived

blood samples. My contributions to this work include the conception and development of DNA patterning techniques, preparation and execution of the hCG experiments, and assistance in the adoption of the plasma separation microfluidics. Chapter 2 is largely derived from © *Nature Biotech.* **2008**, 26(12), 1373-1378.

Chapter 3 describes significant improvements to the quality of barcode microarrays. We found that our initial patterning protocols produced microarrays that yielded consistent measurements over small areas, but varied widely across the substrate as a whole. In order to make valid comparisons amongst single cell data or amongst multiple patient samples analyzed on the same chip, the microarray must present consistent sensitivity throughout. We describe two different strategies that help us to achieve this consistency: one method utilizes covalent attachment of DNA to a modified substrate surface, while a second method preserves the original scheme, but explores the incorporation of DMSO (a common microarraying additive) with the patterning solution. Both strategies yielded barcodes with far better consistency than our initial protocol. A theoretical simulation was undertaken to explain the dramatic improvements achieved by DMSO, and its mechanism was found to differ significantly in microfluidic systems when compared to regular pin spotting. My contribution to this work was the development and characterization of the covalent DNA patterning strategy. Chapter 3 is largely derived from © *ChemPhysChem*. **2010**, 11(14), 3063-3069.

Whereas Chapter 3 focused on improving consistency across single chips, Chapter 4 describes efforts to improve chip-to-chip consistency by automating the flow patterning process. We describe the design and fabrication of a robotics system that is capable of autonomously interfacing with and manipulating microfluidics systems. A modular design philosophy enables it to process almost any flow-through microfluidic substrate with little modification, although we focus on barcode chips. The pattern fidelity of machine-made substrates is confirmed, and both intra- and inter-chip consistency is investigated. Finally, a pair of substrates is used to perform massively parallel single-cell secretion studies of a macrophage cell line, and a statistical analysis of the results demonstrates that data from the two chips are indistinguishable. This chapter is derived from a manuscript that is currently under review.

Finally, Chapter 5 describes an additional set of technologies that were developed to facilitate and automate microfluidics-based experiments in anticipation of clinical trials based on our blood chip. We first develop an intuitive, GUI-based software package that is aimed at laboratory-scale microfluidic control and automation. We also describe the design and fabrication of a self-contained, portable, and modular solenoid array for microfluidic control, and integrate its operation with the aforementioned program. Finally, we discuss the development and basic characterization of a second portable system that optimizes and comprehensively automates microfluidic blood assays such that even non-technical users who have never worked with microfluidics can regularly obtain the same standard of data that is produced in the lab. This latter system is

anticipated to be the basis of upcoming clinical blood trials to characterize blood protein signatures in various cancers.

1.7 Figures

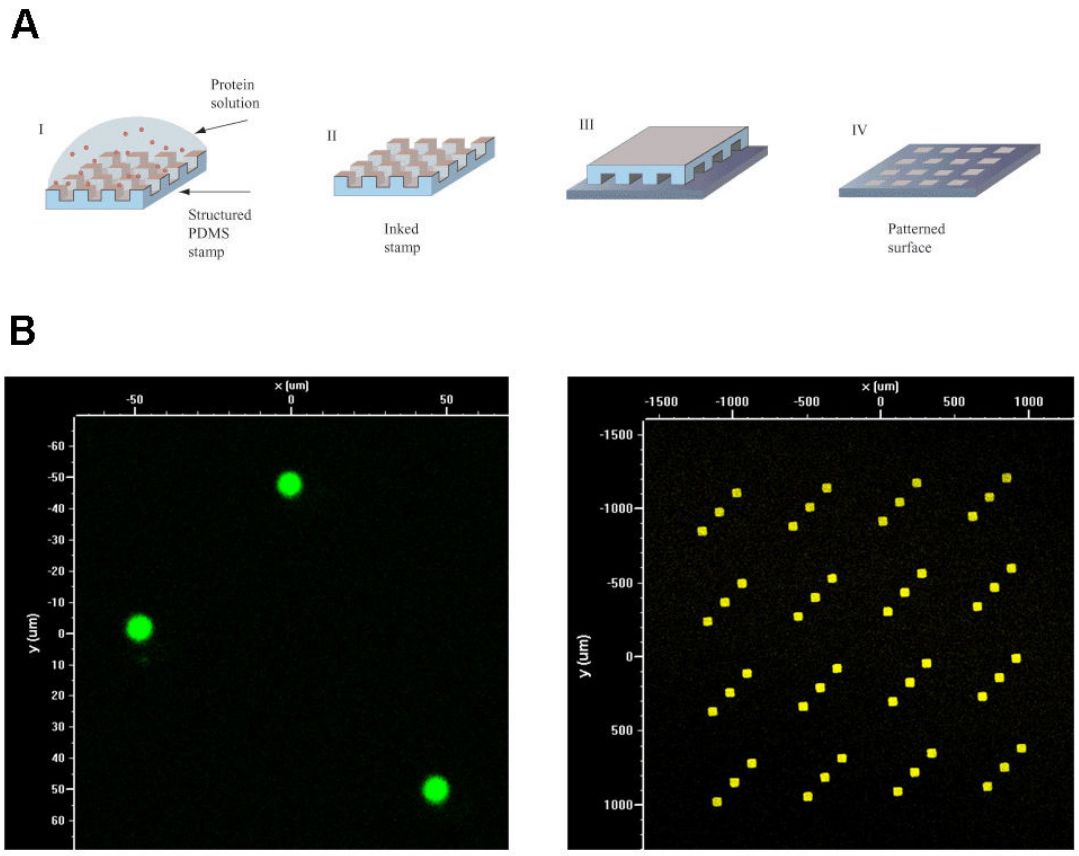


Figure 1.7.1 (A) Schematic illustration of the microcontact printing process. **(B)** Sample stamping results obtained using fluorescent DNA. The circles feature ~5µm diameter while the squares are ~50µm wide.

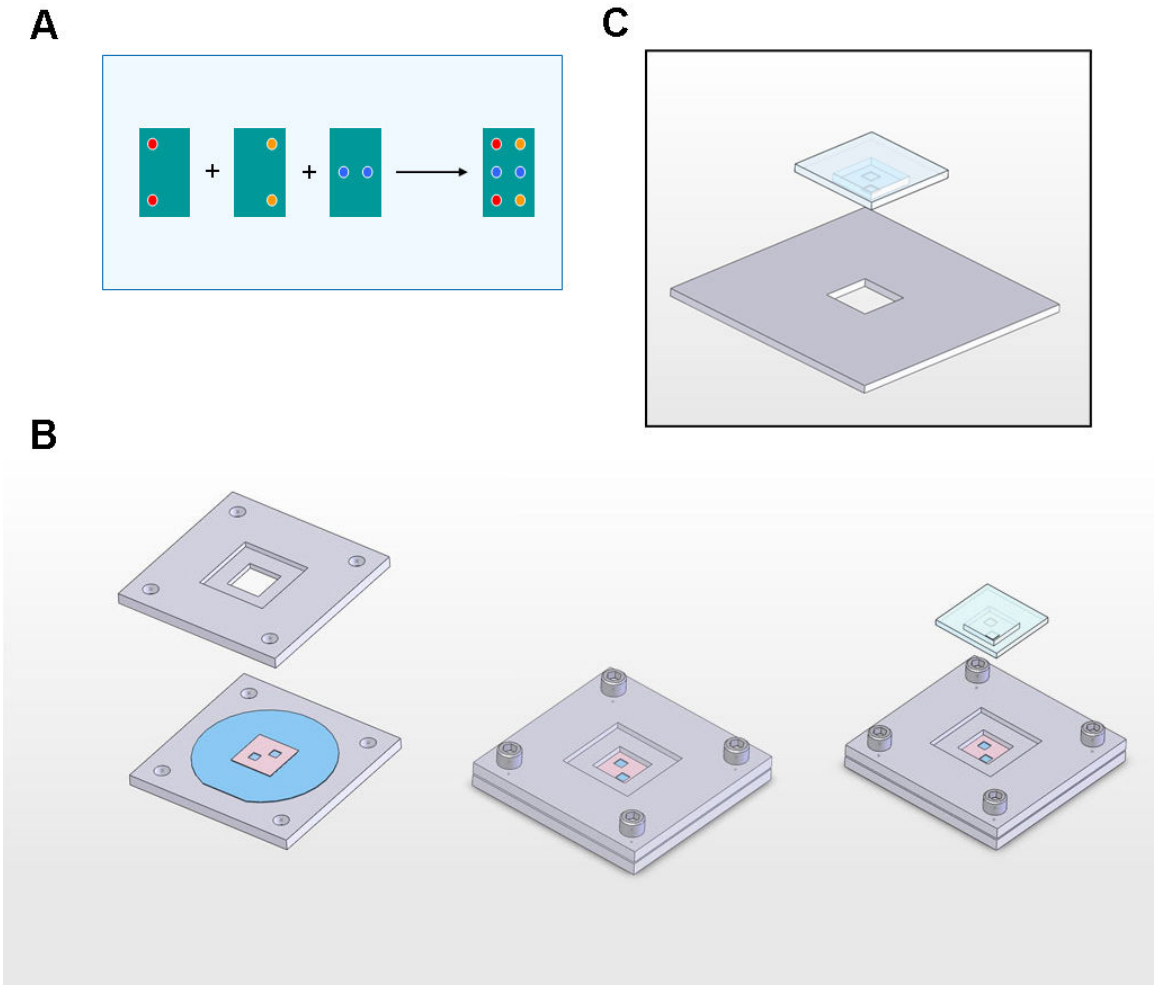


Figure 1.7.2 (A) Conceptual illustration of a microcontact printing scheme to create multi-element DNA microarrays, presuming the stamps can be aligned precisely. **(B)** The molding process by which two-tiered PDMS stamps are fabricated. **(C)** Mask plate with central cutout to accommodate stamp.

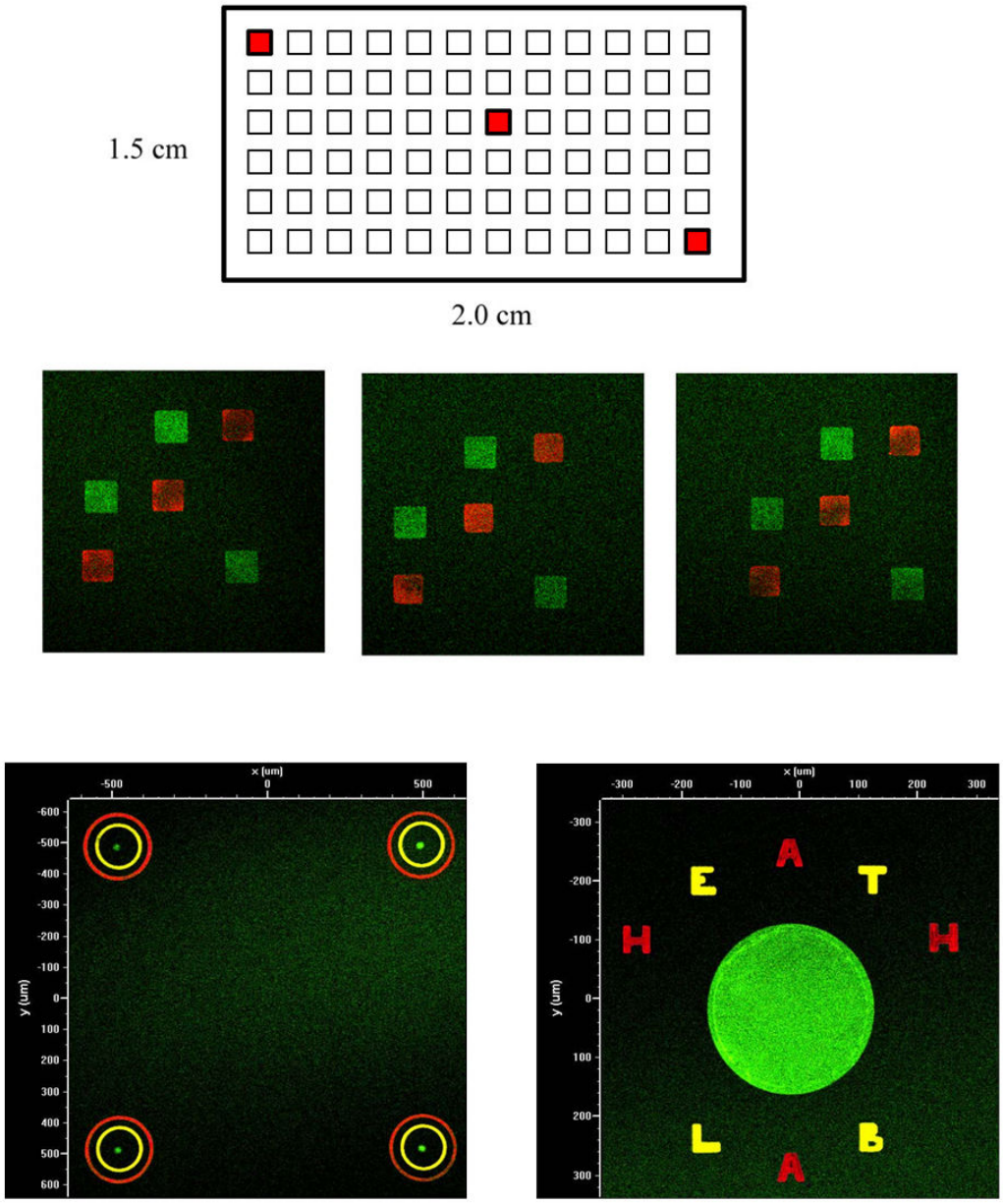


Figure 1.7.3 Typical results from MA-6 based stamp alignment. The middle three patterns correspond to the highlighted boxes in the schematic above, and demonstrate the uniform alignment of two stamps (red & green) across long distances.

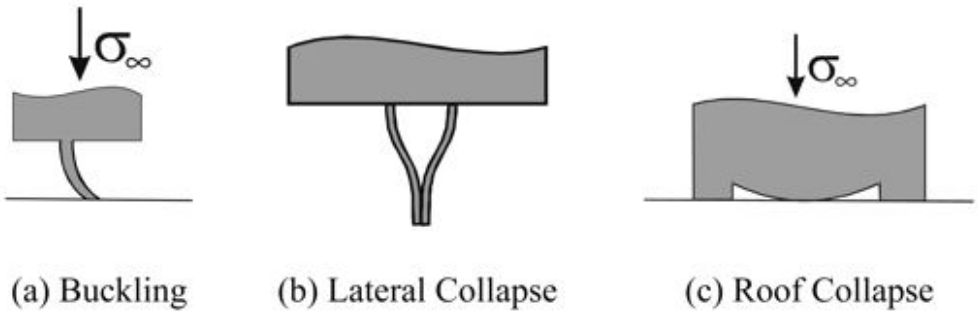


Figure 1.7.4 Common failure modes for high aspect ratio stamp features (**a** & **b**) and low aspect ratio stamps (**c**).

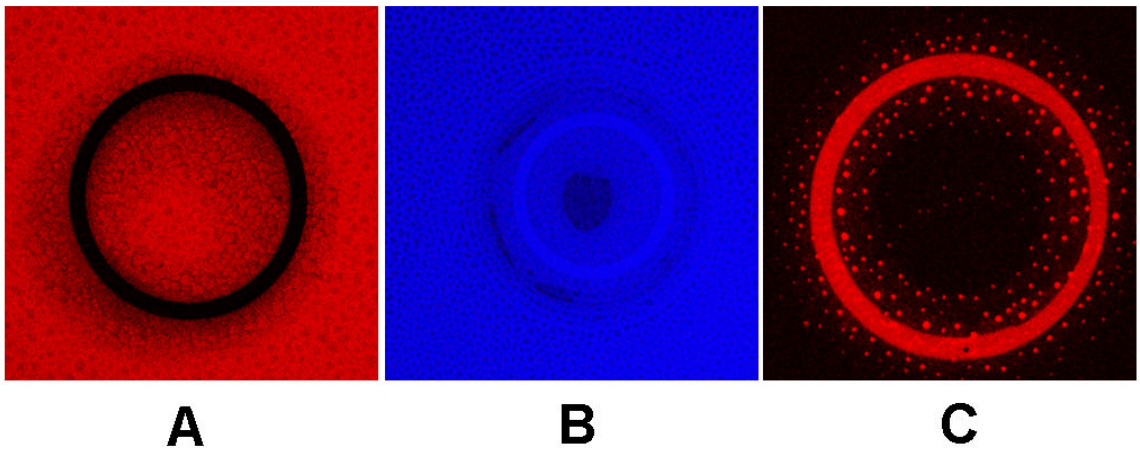


Figure 1.7.5 (A) Fluorescent, DNA-tagged capture antibodies bind nonspecifically to the background, but strongly avoid their complementary strands that are stamped in a ring structure. **(B)** When the substrate is not blocked with BSA, the ring structure becomes evenly populated upon exposure to fluorescent capture antibodies, implying that BSA is preferentially recruited by contaminants that co-deposit with DNA. **(C)** Blocking with a combination of PEG and acetic anhydride prevents non-specific binding to the background, but permits capture antibodies to localize to their complementary sequences.

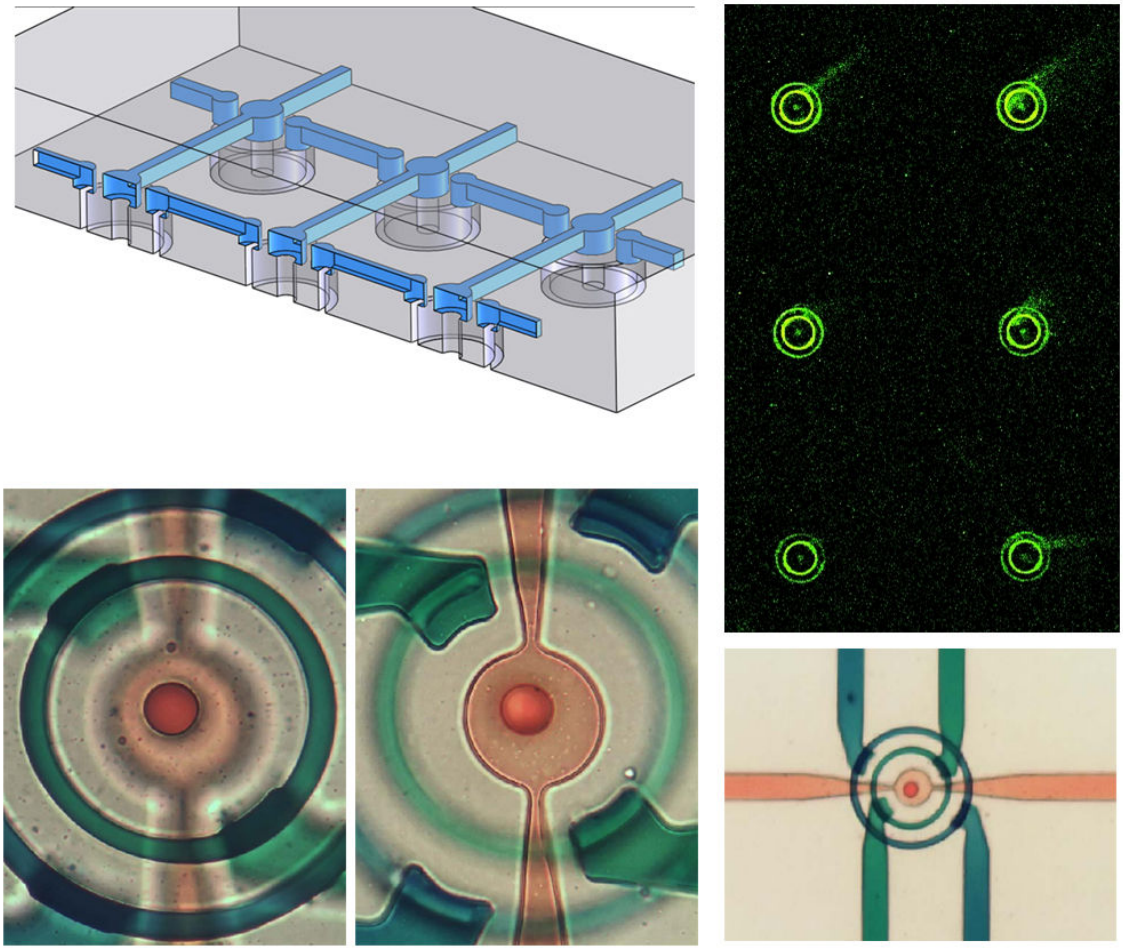


Figure 1.7.6 Counter-clockwise from top left: schematic showing 3D flow channels for patterning discontinuous features; low depth-of-field images showing dye solutions in both lower and upper portions of the flow channels, indicating operational vias; DEAL assays performed with recombinant proteins on a bulls-eye substrate.

1.8 References

1 Strange, K. The end of "naïve reductionism": rise of systems biology or renaissance of physiology? *American Journal of Physiology - Cell Physiology* **288**, C968-C974, (2005).

2 Hood, L., Heath, J. R., Phelps, M. E. & Lin, B. Systems Biology and New Technologies Enable Predictive and Preventative Medicine. *Science* **306**, 640-643, (2004).

3 Weinberg, R. *The Biology of Cancer*. (Garland Sci., 2006).

4 Anderson, N. L. & Anderson, N. G. The human plasma proteome: history, character, and diagnostic prospects. *Molecular and Cellular Proteomics* **1**, 845-867, (2002).

5 Zheng, G., Patolsky, F., Cui, Y., Wang, W. U. & Lieber, C. M. Multiplexed electrical detection of cancer markers with nanowire sensor arrays. *Nat Biotech* **23**, 1294-1301, (2005).

6 Kharitonov, A. B., Wasserman, J., Katz, E. & Willner, I. The Use of Impedance Spectroscopy for the Characterization of Protein-Modified ISFET Devices: Application of the Method for the Analysis of Biorecognition Processes. *The Journal of Physical Chemistry B* **105**, 4205-4213, (2001).

7 Osterfeld, S. J. *et al.* Multiplex protein assays based on real-time magnetic nanotag sensing. *Proceedings of the National Academy of Sciences* **105**, 20637-20640, (2008).

8 Bailey, R. C., Kwong, G. A., Radu, C. G., Witte, O. N. & Heath, J. R. DNA-Encoded Antibody Libraries: A Unified Platform for Multiplexed Cell Sorting and Detection of Genes and Proteins. *Journal of the American Chemical Society* **129**, 1959-1967, (2007).

9 Kumar, A. & Whitesides, G. M. Features of gold having micrometer to centimeter dimensions can be formed through a combination of stamping with an elastomeric stamp and an alkanethiol ``ink" followed by chemical etching. *Applied Physics Letters* **63**, 2002-2004, (1993).

10 Xia, Y. & Whitesides, G. M. SOFT LITHOGRAPHY. *Annual Review of Materials Science* **28**, 153-184, (1998).

11 Lange, S. A., Benes, V., Kern, D. P., Hörber, J. K. H. & Bernard, A. Microcontact Printing of DNA Molecules. *Analytical Chemistry* **76**, 1641-1647, (2004).

12 Thibault, C. *et al.* Direct microcontact printing of oligonucleotides for biochip applications. *Journal of Nanobiotechnology* **3**, 7, (2005).

13 Chakra, E. B., Hannes, B., Dilosquer, G., Mansfield, C. D. & Cabrera, M. A new instrument for automated microcontact printing with stamp load adjustment. *Review of Scientific Instruments* **79**, 064102-064109, (2008).

14 Trinkle, C. A. & Lee, L. P. High-precision microcontact printing of interchangeable stamps using an integrated kinematic coupling. *Lab on a Chip* **11**, 455-459, (2011).

15 Choonee, K. & Syms, R. R. A. Multilevel Self-Aligned Microcontact Printing System. *Langmuir* **26**, 16163-16170, (2010).

16 Nagarah, J. M. *et al.* Batch Fabrication of High-Performance Planar Patch-Clamp Devices in Quartz. *Advanced Materials* **22**, 4622-4627, (2010).
17 Melosh, N. A. *et al.* Ultrahigh-Density Nanowire Lattices and Circuits. *Science* **300**, 112-115, (2003).
18 Sharp, K. G., Blackman, G. S., Glassmaker, N. J., Jagota, A. & Hui, C.-Y. Effect of Stamp Deformation on the Quality of Microcontact Printing: Theory and Experiment. *Langmuir* **20**, 6430-6438, (2004).
19 Hui, C. Y., Jagota, A., Lin, Y. Y. & Kramer, E. J. Constraints on Microcontact Printing Imposed by Stamp Deformation. *Langmuir* **18**, 1394-1407, (2002).
20 Schmid, H. & Michel, B. Siloxane Polymers for High-Resolution, High-Accuracy Soft Lithography. *Macromolecules* **33**, 3042-3049, (2000).
21 Choi, K. M. & Rogers, J. A. A Photocurable Poly(dimethylsiloxane) Chemistry Designed for Soft Lithographic Molding and Printing in the Nanometer Regime. *Journal of the American Chemical Society* **125**, 4060-4061, (2003).
22 Odom, T. W., Love, J. C., Wolfe, D. B., Paul, K. E. & Whitesides, G. M. Improved Pattern Transfer in Soft Lithography Using Composite Stamps. *Langmuir* **18**, 5314-5320, (2002).
23 Böhm, I., Lampert, A., Buck, M., Eisert, F. & Grunze, M. A spectroscopic study of thiol layers prepared by contact printing. *Applied Surface Science* **141**, 237-243, (1999).
24 Glasmästar, K., Gold, J., Andersson, A.-S., Sutherland, D. S. & Kasemo, B. Silicone Transfer during Microcontact Printing. *Langmuir* **19**, 5475-5483, (2003).
25 Sharpe, R. B. A. *et al.* Ink Dependence of Poly(dimethylsiloxane) Contamination in Microcontact Printing. *Langmuir* **22**, 5945-5951, (2006).
26 Lee, J. N., Park, C. & Whitesides, G. M. Solvent Compatibility of Poly(dimethylsiloxane)-Based Microfluidic Devices. *Analytical Chemistry* **75**, 6544-6554, (2003).
27 Cheung, V. G. *et al.* Making and reading microarrays. *Nat Genet*.
28 Anderson, J. R. *et al.* Fabrication of Topologically Complex Three-Dimensional Microfluidic Systems in PDMS by Rapid Prototyping. *Analytical Chemistry* **72**, 3158-3164, (2000).
29 David, J. & et al. Soft and rigid two-level microfluidic networks for patterning surfaces. *Journal of Micromechanics and Microengineering* **11**, 532, (2001).
30 Luo, Y. & Zare, R. N. Perforated membrane method for fabricating three-dimensional polydimethylsiloxane microfluidic devices. *Lab on a Chip* **8**, 1688-1694, (2008).

Chapter 2

Integrated barcode chips for rapid, multiplexed analysis of proteins in microliter quantities of blood

2.1 Introduction

As the tissue that contains the largest representation of the human proteome[1], blood is the most important fluid for clinical diagnostics[2-4]. However, although changes of plasma protein profiles reflect physiological or pathological conditions associated with many human diseases, only a handful of plasma proteins are routinely used in clinical tests. Reasons for this include the intrinsic complexity of the plasma proteome[1], the heterogeneity of human diseases and the rapid degradation of proteins in sampled blood[5]. We report an integrated microfluidic system, the integrated blood barcode chip that can sensitively sample a large panel of protein biomarkers over broad concentration ranges and within 10 min of sample collection. It enables on-chip blood separation and rapid measurement of a panel of plasma proteins from quantities of whole blood as small as those obtained by a finger prick. Our device holds potential for

inexpensive, noninvasive and informative clinical diagnoses, particularly in point-of-care settings.

Microfluidics has permitted the miniaturization of conventional techniques to enable high-throughput and low-cost measurements in basic research and clinical applications[6,7]. Systems for biomolecular assays[8,9] and bio-separations[10,11], including the separation of circulating tumor cells or plasma from whole blood[12-14], have been reported. We developed the integrated blood barcode chip (IBBC) to address the need for microchips that integrate on-chip plasma separations from microliter quantities of whole blood with rapid in situ measurements of multiple plasma proteins. The immunoassay region of the chip is a microscopic barcode, integrated into a microfluidics channel and customized for the detection of many proteins and/or for the quantification of a single or few proteins over a broad concentration range. We demonstrate versatility of this barcode immunoassay by detecting human chorionic gonadotropin (hCG) from human serum over a 105 concentration range and by stratifying 22 cancer patients via multiple measurements of a dozen blood protein biomarkers for each patient. We also use the IBBC to assay a blood protein biomarker panel from whole human blood, performing all key steps in the immunoassay within 10 min of blood collection by finger prick.

2.2 Results and Discussion

2.2.1 Device design

We first present an overview of the IBBC and then discuss control of assay sensitivity, extension of a single protein assay to an assay for a large panel of biomarkers and, finally, integration of plasma separation from whole blood, followed by the rapid measurement of a panel of protein biomarkers. Figure 2.5.1 shows the design of an IBBC for blood separation and in situ protein measurement. We designed a polydimethylsiloxane (PDMS)-on-glass chip to perform 8–12 separate multiprotein assays sequentially or in parallel, starting from whole blood.

The Zweifach-Fung effect describes highly polarized blood cell flow at branch points of small blood vessels[14-16]. A component of the IBBC, redesigned from a previous report[14], exploits this hydrodynamic effect by flowing blood through a low-flow-resistance primary channel with high-resistance, centimeter-long channels that branch off it at right angles (Figure 2.5.1a). As the resistance ratio is increased between the branches and the primary channel, a critical streamline moves closer to the primary channel wall adjoining the branch channels. Blood cells with a radius larger than the distance between this critical streamline and the primary channel wall are directed away from the high-resistance channels, and ~15% of the plasma is skimmed into the high-resistance channels. The remaining whole blood is directed toward a waste outlet. The glass base of the plasma-skimming channels is patterned with a dense barcode-like

array of single-stranded DNA (ssDNA) oligomers before assembly of the microfluidics chip. A full barcode is repeated multiple times within a single plasma-skimming channel, and each barcode sequence constitutes a complete assay.

We used the DNA-encoded antibody library (DEAL) technique[17] to detect proteins within the plasma-skimming channels. DEAL technology involves using DNA-directed immobilization of antibodies to convert a prepatterned ssDNA barcode microarray into an antibody microarray, thereby providing a powerful means for spatial encoding[18,19]. The sequences of all ssDNA oligomer pairs used (labeled A/A'-M/M'), and their corresponding antibodies, are listed in Tables 2.6.1 and 2.6.2. To minimize cross-reactivity, these ssDNA molecules were designed in silico and then validated through a full orthogonality test (Figure 2.5.2). In that experiment, each of the complementary DNA molecules with Cy3 fluorescent label was added to a microwell containing a full primary ssDNA barcode array. The results showed only negligible cross-hybridization signals. In the DEAL assay, each capture antibody is tagged with approximately three copies of an ssDNA oligomer that is complementary to ssDNA oligomers that have been surface-patterned into a microscopic barcode within the immunoassay region of the chip. Flow-through of the DNA-antibody conjugates transforms the DNA microarray into an antibody microarray for the subsequent surface-bound immunoassay. Because DNA patterns are robust to dehydration and can survive elevated temperatures (80–100 °C), the DEAL approach circumvents the denaturation of antibodies often associated with typical microfluidics fabrication.

As only a few microliters of blood is normally sampled from a finger prick, on-chip plasma separation yields only a few hundred nanoliters of plasma. The ssDNA barcodes were patterned at a high density using microchannel-guided flow patterning (Figure 2.5.3) to measure a large panel of protein biomarkers from this small volume. We used a PDMS mold that was thermally bonded onto a polyamine-coated glass slide to pattern the entire ssDNA barcode. Polyaminated surfaces permit substantially higher DNA loading than do more traditional aminated surfaces[20] and provide for an accompanying increase in assay sensitivity (Figures 2.5.4 and 2.5.5). Different solutions, each containing a specific ssDNA oligomer, were flowed through different channels and evaporated through the gas-permeable PDMS stamp, resulting in individual stripes of DNA molecules. One complete set of stripes represents one barcode. All measurements used 20-mm-wide bars spaced at a 40 mm pitch. This array density represents an approximately tenfold increase over a standard spotted array (typical dimensions are 150 mm diameter spots at a 400 mm pitch), thus expanding the numbers of proteins that can be measured within a small volume. No alignment between the barcode array and the plasma channels was required. All protein assays used one color fluorophore and were spatially identified using a reference marker that fluoresced at a different color.

2.2.2 Assay sensitivity as a function of DNA patterning concentration

We first illustrate aspects of the barcode assays via the measurement of a single biomarker, human chorionic gonadotropin (hCG), in undiluted human serum over a broad concentration range. hCG is widely used for pregnancy testing and is a biomarker for gestational trophoblastic tumors and germ cell cancers of the ovaries and testes. For this assay, the barcode was customized by varying the DNA loading during the flow patterning step. The DNA barcode contained 13 regions (Figure 2.5.6a). There were two bars of oligomer B (designed to detect the protein tumor necrosis factor (TNF)-α as a negative control), one reference bar (oligomer M), one blank and nine bars of oligomer A (designed for hCG detection and flow patterned at ssDNA concentrations that were varied from 200 μM to 2 μM). To perform the assay, we flowed a mixture of A'-anti-hCG and B'-TNF- α through assay channels. Next, a series of standard hCG serum samples and two hCG samples of unknown concentration were flowed through separate assay channels. Biotinylated detection antibodies for hCG and TNF-a were then applied, followed by a final developing step using fluorescent Cy5-labeled streptavidin (red) for all protein channels and Cy3-labeled M' oligomers (green) for the reference channel (Figure 2.5.6a). Quantifying the fluorescence intensity (Figure 2.5.6b,c) revealed a sensitivity (~1 mIU/ml) comparable to the enzyme-linked immunosorbent assays (ELISAs) over a broad detected concentration range (~10^5). Using the microfluidics-entrained DEAL barcode in a blind test, we measured the hCG levels in the two unknown serum samples. Our measured levels, estimated at 6 and 400 mIU/ml for unknowns 1 and 2, are in good agreement with the values of 12 and 357 mIU/ml, respectively,

obtained from an independent lab test. Even without quantification, the analyte concentrations can be estimated by eye through pattern recognition of the full barcode. The bar with the highest DNA-loading rendered the highest sensitivity, whereas the bar with lowest DNA-loading was used to discriminate samples with high analyte concentrations. For example, the 25,000 mIU/ml and 250 mIU/ml hCG samples can be visually distinguished using stripes patterned with lower DNA concentrations, whereas the stripes loaded from 200 mm DNA solutions do not readily distinguish these samples. For circumstances in which accurate photon counting is not available, visual barcode inspection permits a rough estimation of the target quantity—a potential point-of-care application. When levels of hCG are tracked during pregnancy, concentrations in the blood increase from ~5 mIU/ml in the first week of pregnancy to ~2 x 10^5 mIU/ml 10 weeks after conception. The IBBC can cover such a broad physiological hCG range with reasonable accuracy.

To evaluate multiplexed measurements of a panel of 12 protein markers using the microfluidic DEAL barcode regions of the IBBCs, we quantified the cross-reactivity between the stripes within the DNA-encoded immunoassays. This test involved twelve human serum proteins, including ten cytokines (interferon (IFN)-γ, TNF-α, interleukin (IL)-2, IL-1α, IL-1β, transforming growth factor (TGF)-β1, IL-6, IL-10, IL-12, granulocyte-macrophage colony-stimulating factor (GM-CSF)), a chemokine macrophage chemoattractant protein (MCP)-1 and the cancer biomarker prostate-specific antigen (PSA). The results showed negligible cross-talk, with typical photon counts

<2% compared to the correctly paired antigen-antibody complexes (Figure 2.5.7). We also assayed serial dilutions (from 5 nM to 1 pM) for these proteins on the DEAL barcode chip to establish a set of calibration curves for future estimates of protein concentration in sera (Figure 2.5.8). We fixed all the parameters associated with laser scanning and fluorescence quantification (e.g., power, gain, brightness and contrast) and performed quantitative analysis. Depending on the antibodies used, the estimated sensitivity varies from <1 pM for IL-1β and IL-12 to ~30 pM for TGF-β and is comparable to the detection limits of ELISAs based on the same antibody pairs. For example, according to the specifications of commercial kits (eBioscience), the detection limit for cytokines like TNF-a and IL-1b is B8 pg/ml (~0.5 pM), which compares favorably with our observations. However, the statistical variation of the measured signals is relatively large compared to a commercial ELISA assay—a variation that likely arises from our manual chip manufacturing.

2.2.3 Multi-parameter analysis of frozen serum samples

We assessed the utility of the DEAL barcodes for clinical blood samples by measuring the same 12 proteins from small amounts of stored serum collected from 22 cancer patients. These serum samples were thawed, and then assayed using two chips, each containing 12 separate assay units operated in parallel. In every unit, 20 full DEAL barcodes in each assay channel were used for statistical sampling. The proteins in this panel (Fig. 2.5.9a), the prostate cancer marker PSA and eleven proteins secreted by various white blood cells, have been associated with tumor microenvironment

formation, tumor progression and tumor metastasis[21-23]. Thus, this panel provides information relevant to multiple aspects of cancer.

Figure 2.5.9b shows fluorescence images, each depicting four sets of randomly picked barcodes obtained from the 22 patient samples. The medical records for all patients are summarized in Table 2.6.3. B01–B11 denote 11 samples from breast cancer patients, whereas P01–P11 are from prostate cancer patients. Many proteins were successfully detected with high signal-to-noise ratios, and the barcode signatures are distinctive from patient to patient, excepting the assays on P05, P04, P10 and B10. These assays are from individuals who are heavy smokers (~11–20 cigarettes daily). Only one serum sample (P06) from a heavy smoker did not exhibit a high background. This high background may result from elevated blood content of the fluorescent protein carboxyhemoglobin, which has been shown relevant to the pathogenesis of lung diseases of smokers[24]. Although we have also measured high background in a number of stored serum samples, we have never measured a high background in assays from very freshly collected blood, as described below. The results imply that, at least for stored samples, some prepurification of the plasma or serum will be required to assay serum protein levels.

Barcode intensities were then quantified and the statistic mean value for each protein was computed. The cancer marker PSA clearly distinguished between the breast cancer and the prostate cancer patients. The only exception was a false-positive result from

B10 that had high nonspecific background. We independently validated our PSA measurements using the standard ELISA for PSA in all patient sera. For eight of the prostate cancer patients, we compared these results with clinical ELISA measurements provided by the serum supplier. The results (Fig. 2.5.9c) validated the applicability of the DEAL barcodes for assaying complex clinical samples. However, the statistical accuracy of the PSA barcode assay was not high, revealing only a modest linear correlation between the ELISA and DEAL. Again, this is likely due to our manual chip manufacturing process. We are currently automating our barcode fabrication, assay execution and image quantification in an effort to bring statistical uncertainties to within 10–20%, which would be close to the state of the art.

The cancer patient barcode data could be analyzed for absolute protein levels by comparing those data against the barcode quantification plots (Figure 2.5.8). Results for PSA, TNF-a and IL-1b are shown in Figure 2.5.9d. PSA concentrations range from 22 pM to 1 nM (or 0.7 to 33 ng/ml) with a log-scale mean of 117 pM (3.8 ng/ml) for prostate cancer patients. The estimated PSA concentrations for breast cancer patient sera has a mean of 9.1 pM. PSA readily differentiates between these two patient groups with good statistical accuracy ($P = 0.0007$). Nevertheless, the absolute PSA levels measured by either the standard ELISA or by the barcode assay are below those determined by the clinical ELISA—a likely result of sample degradation during storage (Figure 2.5.9c). As would be expected, neither TNF-α nor IL-1β allows prostate and breast cancer patients to be distinguished ($P = 0.4$ and 0.5, respectively at significance level 0.2). Our estimates

of absolute protein levels indicate that the protein concentration ranges assessed by the DEAL barcode assay are clinically relevant for patient diagnostics. For example, the serum level of cytokines such as interleukins and tumor necrosis factors can reach ~10–100 pg/ml in cancer patients[25], ~500 pg/ml in rheumatoid arthritis patients and 41ng/ml[26] in septic shock[27]. These levels can all be captured using the barcode assay format.

We performed a complete nonsupervised clustering (that is, using only the levels of assayed proteins without assigning any weight factors) of patients and generated the heat map (Fig. 2.5.9e) to assess the potential of this technology for patient stratification. This analysis is only presented as a proof of principle. Nevertheless, the results are encouraging. For example, the measured profiles of breast cancer patients can be classified into three subsets—noninflammatory, IL-1β positive and TNF-α/GMCSF positive ($P_{TNF\alpha}$ = 0.005, P_{GMCSF} = 0.04 for the latter two subsets). The prostate cancer patient data were classified into two major subsets based upon the inflammatory protein levels ($P_{TNF\alpha}$ = 0.016, P_{GMCSF} = 0.012). The multiplexed measurement of cytokines[28] is relevant to cancer diagnostics and prognostics[29,30]. Our results demonstrate that IBBCs can be applied to the multiparameter analysis of human health-relevant proteins in serum.

2.2.4 Multi-parameter fresh blood analysis

The ultimate goal behind developing the IBBC was to measure the levels of a large number of proteins in human blood within a few minutes of sampling that blood, to avoid the protein degradation that can occur when plasma is stored. In a typical 96-well plate immunoassay, the biological sample of interest is added, and the protein diffuses to the surface-bound antibody. Under adequate flow conditions, diffusion is no longer important, and the only parameter that limits the speed of the assay is the protein/antibody binding kinetics (the Langmuir isotherm)[31], thus allowing the immunoassay to be completed in just a few minutes[32]. Flow through our plasma-skimming channels proceeds at velocities >~0.1 mm sec^{-1} and can operate continuously and with near 100% efficiency unless the blood flow is clogged.

For whole blood analysis, the microfluidic channels of IBBCs were precoated with bovine serum albumin blocking buffer. The DNA barcodes were transformed into antibody barcodes as described above, and blood samples were flowed into the device within 1 min of fingerprick collection. The time from that fingerprick to completion of blood flow through the device was ~9 min. We sampled both as collected whole blood and protein-spiked blood from healthy volunteers. Figure 2.5.10a shows the effective separation of plasma in an IBBC. The few red blood cells that did enter the plasma channels (Figure 2.5.10a, right panel) did not affect the subsequent protein assay.

The plasma proteins detected in this whole-blood analysis experiment included a cancer marker (PSA), four cytokines and three other functional proteins (complement C3, C-reactive protein (CRP) and plasminogen) involved in the complement system, inflammatory response, fibrin degradation and liver toxicity (Tables 2.6.1 and 2.6.2). After exposure of the barcode assay region to the separated, flowing plasma for 8 min, the detection antibody solution and the fluorescence probes were added to complete the assay. All proteins in the spiked blood were detected (Fig. 2.5.10b,c). Cytokines gave the strongest fluorescence signals because of higher affinities of their cognate antibodies. The measurement of the unspiked fresh blood established a baseline for a healthy volunteer, in which IL-6, IL-10, C3 and plasminogen were detected. Using IBBCs for the separation and analysis of very freshly collected blood consistently resulted in very clean DEAL barcodes, with little or no evidence of biofouling. We are planning a study to assess the importance of rapid measurements for obtaining accurate protein levels.

2.3 Conclusions

Our IBBC enables the rapid measurement of a panel of plasma proteins from a finger prick of whole blood. Integration of microfluidics and DNA-encoded antibody arrays enables reliable processing of blood and in situ measurement of plasma proteins within a time scale that is short enough to avoid most protein degradation processes that can

occur in sampled blood. Use of the IBBC represents a minimally invasive, low-cost and robust procedure, and potentially represents a realistic clinical diagnostic platform.

2.4 Experimental Methods

2.4.1 Micropatterning of barcode array.

A PDMS mold containing 13–20 parallel microfluidic channels, with each channel conveying a different DNA oligomer as DEAL code, was fabricated by soft lithography. The PDMS mold was bonded to a polylysine-coated glass slide via thermal treatment at 80 °C for 2 hours. The polyamine surfaces permit significantly higher DNA loading than do more traditional aminated surfaces. DNA 'bars' of 2 µm in width have been successfully patterned using this technique. In the present study, a 20-µm channel width was chosen because the fluorescence microarray scanner we used has a resolution of 5 µm. Nevertheless, the current design already resulted in a DNA barcode array an order of magnitude denser than conventional microarrays fabricated by pin-spotting. The coding DNA solutions (A-M for the cancer serum test and AA-HH for the finger-prick blood test) prepared in 1x PBS were flowed into individual channels, and then allowed to evaporate completely. Finally, the PDMS was peeled off and the substrate with DNA barcode arrays was baked at 80 °C for 2–4 hours. The DNA solution concentration was ~100 µM in all experiments except in the hCG test, leading to a high loading of ~6 x 10^{13} molecules/cm^2 (assuming 50% was collected onto substrate).

2.4.2 Fabrication of IBBCs.

The fabrication of PDMS devices for the IBBCs was accomplished through a two-layer soft lithography approach. The control layer was molded from a SU8 2010 negative photoresist (~20 mm in thickness) silicon master using a mixture of GE RTV 615 PDMS prepolymer part A and part B (5:1). The flow layer was fabricated by spin-casting the pre-polymer of GE RTV 615 PDMS part A and part B (20:1) onto a SPR 220 positive photoresist master at 2,000 r.p.m. for 1 min. The SPR 220 mold was ~17 mm in height after rounding by thermal treatment. The control layer PDMS chip was then carefully aligned and placed onto the flow layer, which was still situated on its silicon master, and an additional 60 min thermal treatment at 80 °C was performed to enable bonding. Afterward, this two-layer PDMS chip was cut off the flow layer master and access holes were punched. Finally, the two-layer PDMS chip was thermally bonded onto the barcode-patterned glass slide, yielding a completed integrated blood barcode chip (IBBC). In this chip, the DEAL barcode stripes are oriented perpendicular to the microfluidic assay channels. Typically, 8–12 identical units were integrated in a single chip with the dimensions of 2.5 cm x 7cm.

2.4.3 Clinical specimens of cancer patient sera.

The stored serum samples from 11 breast cancer patients (all female) and 11 prostate cancer patients (all male) were acquired from Asterand. Nineteen out of 22 patients

were European-American and the remaining three were Asian, Hispanic and African-American. The medical history is summarized in Supplementary Table 3.

2.4.4 Collecting a finger prick of blood.

The human whole blood was collected according to the protocol approved by the institutional review board of the California Institute of Technology. Finger pricks were performed using BD microtainer contact-activated lancets. Blood was collected with SAFE-T-FILL capillary blood collection tubes (RAM Scientific), which we prefilled with 80 µl of 25 mM EDTA solution. A 10 µl volume of fresh human blood from a healthy volunteer was collected in an EDTA-coated capillary, dispensed into the tube, and rapidly mixed by inverting a few times. The spiked blood sample was prepared in a similar way except that 40 µl of 25mM EDTA solution and

40 µl of recombinant solution were mixed and pre-added in the collection tube. Then 2 µl of 0.5 M EDTA was added to bring the total EDTA concentration up to 25 mM.

Execution of blood separation and plasma protein measurement using IBBCs. The IBBCs were first blocked with the buffer solution for 30–60 min. The buffer solution prepared was 1% wt/vol bovine serum albumin fraction V (Sigma) in 150 mM 1x PBS without calcium/magnesium salts (Irvine Scientific). The fluid loading was conducted using a Tygon plastic tubing that is interfaced to the IBBC inlet with a 23 gauge metal pin. The Fluidigm solenoid unit was exploited to control the pressure on/off for both control valves and flow channels. A pressure of 8–10 p.s.i. was applied to actuate the valves,

whereas the loading of fluid into assay channels was carried out with a lower pressure (0.5–3 p.s.i.) depending on the channel flow resistance and the desired flow rate. Then DNA-antibody conjugates (~50–100 nM) were flowed through the plasma assay channels for ~30–45 min. This step transformed the DNA arrays into capture-antibody arrays. Unbound conjugates were washed off by flowing buffer solution through the channels. At this step, the IBBC was ready for the blood test. Two blood samples prepared as mentioned above were flowed into the IBBCs within 1 min of collection. The IBBC quickly separated plasma from whole blood, and the plasma proteins of interest were captured in the assay zone where DEAL barcode arrays were placed. This whole process from finger-prick to plasma protein capture took <10 min. In the cancer-patient serum experiment, the as-received serum samples were flowed into IBBCs without any pre-treatment (that is, no purification or dilution). Afterwards, a mixture of biotin-labeled detection antibodies (~50–100 nM) for the entire protein panel and the fluorescence Cy5-streptavidin conjugates (~100 nM) were flowed sequentially into IBBCs to complete the DEAL immunoassay. The unbound fluorescence probes were rinsed off by flowing the buffer solution for 10 min. At last, the PDMS chip was removed from the glass slide. The slide was immediately rinsed in 1/2x PBS solution and deionized water and then dried with a nitrogen gun. Finally, the DEAL barcode slide was scanned by a microarray scanner.

2.4.5 Quantification and statistics.

All the barcode array slides used in quantification were scanned using an Axon Genepix 4000B two-color laser microarray scanner at the same instrumental settings—100% and 33% for the laser power of 635 nm and 532 nm, respectively. Optical gains are 800 and 700 for 635 nm and 532 nm, respectively. The brightness and contrast were set at 87 and 88. The output JPEG images were carefully skewed and resized to fit the standard mask design of barcode array. Then, an image processing software, NIH imageJ, was used to produce intensity line profiles of barcodes in all assay channels. Finally, all the line profile data files were loaded into a home-developed program embedded as an Excel macro to generate a spreadsheet that lists the average intensities of all 13 bars in each of 20 barcodes. The means and standard divisions were computed using the Microcal origin. Nonsupervised clustering of patients was performed using the literature methods and algorithms[33]. To assess the significance of two patient (sub)groups, Student *t* analysis was performed on selected proteins and all *P*-values were calculated at a significance level of 0.05, if not otherwise specified.

2.5 Figures

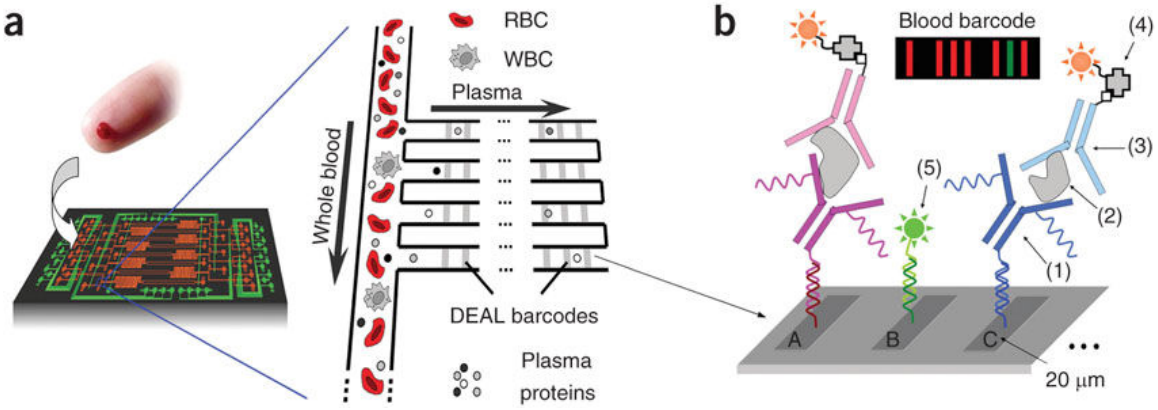


Figure 2.5.1 (a) Scheme depicting plasma separation from a finger prick of blood by harnessing the Zweifach-Fung effect. Multiple DNA-encoded antibody barcode arrays are patterned within the plasma-skimming channels for in situ protein measurements. **(b)** DEAL barcode arrays patterned in plasma channels for *in situ* protein measurement. A, B, C indicate different DNA codes. (1)–(5) denote DNA-antibody conjugate, plasma protein, biotin-labeled detection antibody, streptavidin-Cy5 fluorescence probe and complementary DNA-Cy3 reference probe, respectively. The inset represents a barcode of protein biomarkers, which is read out using fluorescence detection. The green bar represents an alignment marker.

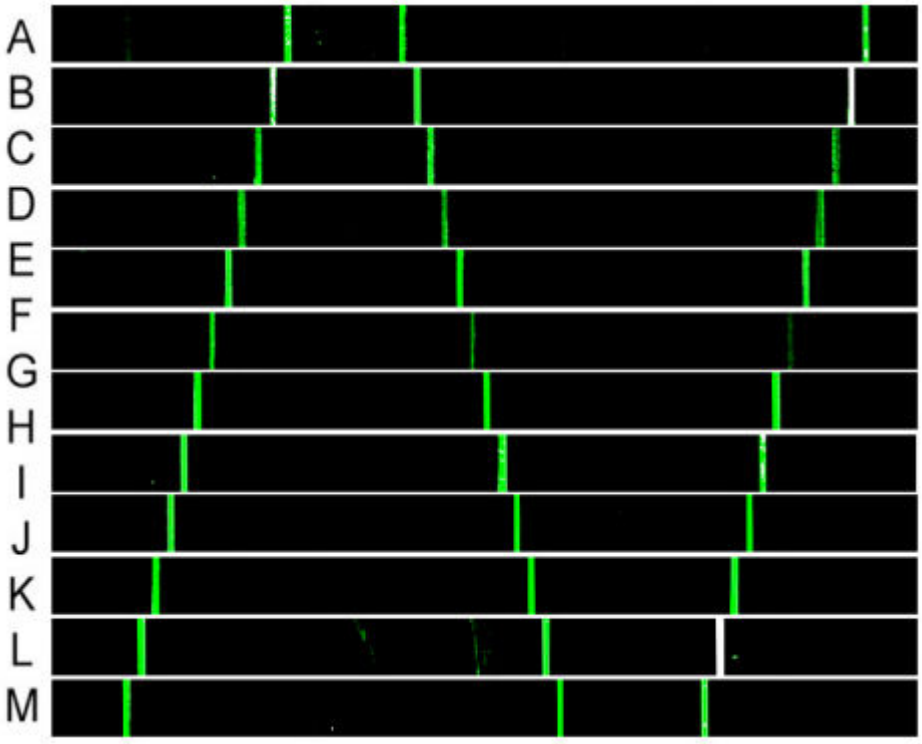


Figure 2.5.2 Cross-hybridization check for all 13 DNA oligomer pairs that were used for encoding the registry of antibody barcode arrays.

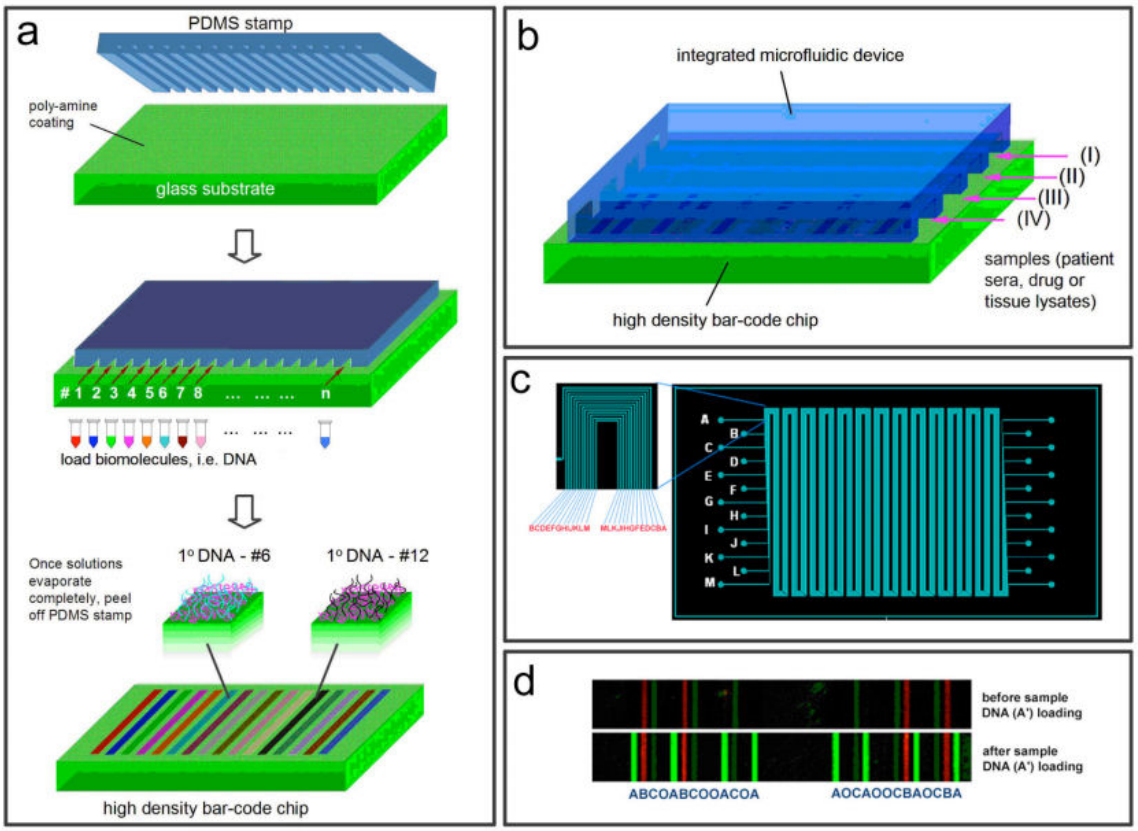


Figure 2.5.3 Microchannel-guided flow patterning of DEAL barcode arrays. **(a)** Depiction of the procedure. Each DNA bar is 20 µm wide and spans the dimensions of the glass substrate. **(b)** Integration of a DEAL barcode-patterned glass slide with microfluidics for multiplexed protein assays. **(c)** Mask design of a 13-channel barcode. A-M denotes the flow channels for the different DNA molecules. **(d)** Validation of successful patterning of DNA molecules by specific hybridization of oligomer A to its fluorescent complementary strand A'. The primary strands B and C were pre-tagged with red and green dyes as references.

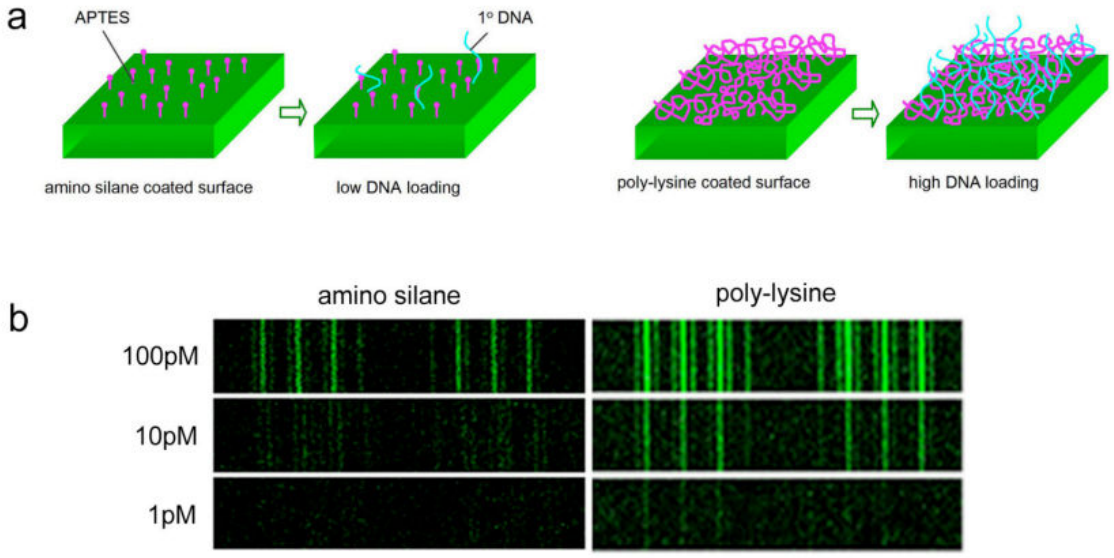


Figure 2.5.4 Effects of polylysine coating on DEAL assay. **(a)** Schematic illustration of polylysine coating for increased loading of DNA oligomer codes. **(b)** Fluorescence images showing a comparative study of the measurement of three human cytokines (IFN-γ, TNF-α and IL-2) using substrates coated with amino-silane and polylysine, respectively.

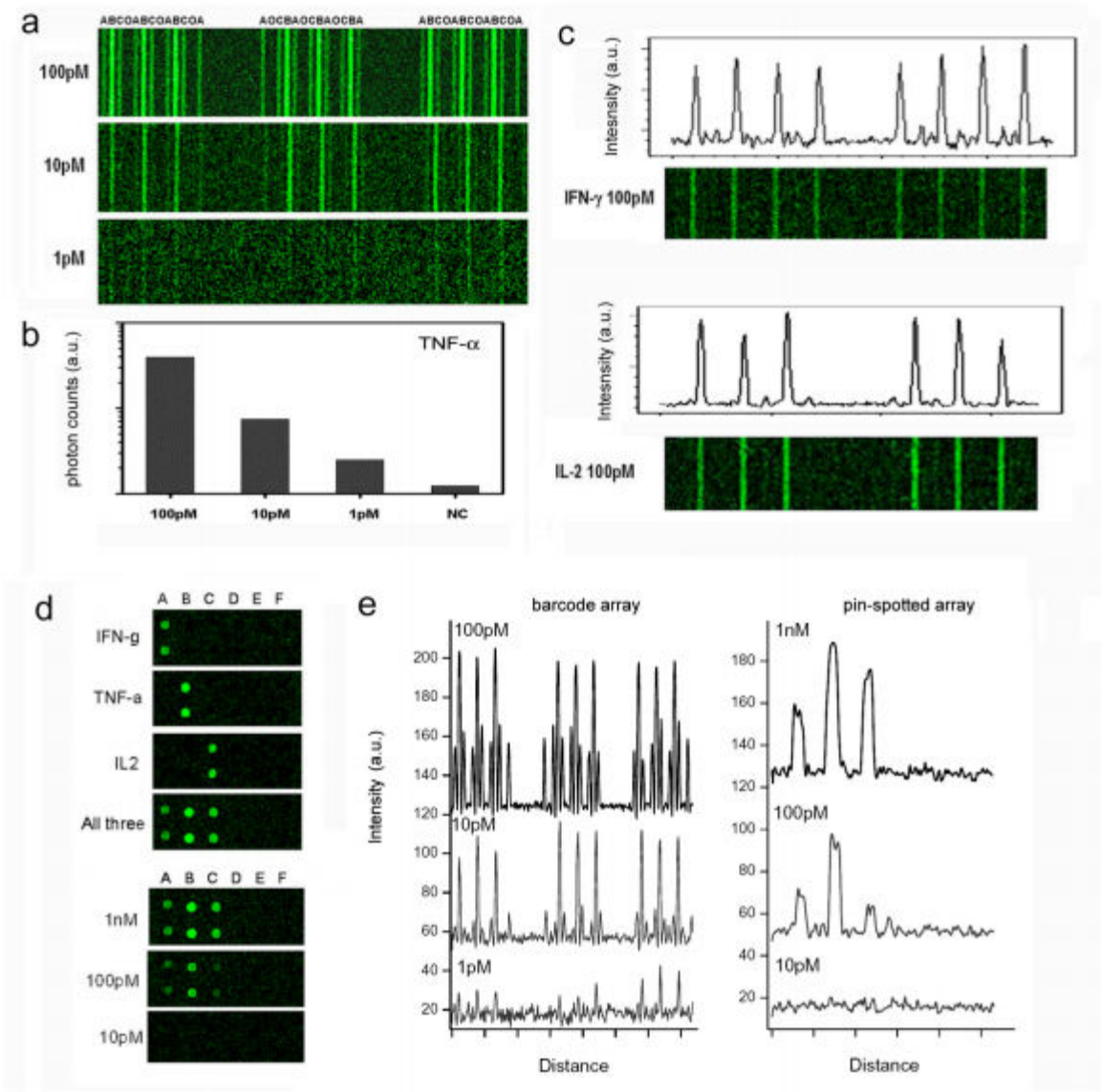


Figure 2.5.5 Increased sensitivity observed in immunoassays run on DEAL barcode arrays. **(a)** Concentration-dependent fluorescence signal for the detection of three human cytokines (A: IFN-γ, B: TNF-α, C: IL-2, O: negative control) using a DEAL barcode array. The bar width is 20 µm. **(b)** Quantitation of fluorescence intensity vs. TNF-α concentration. **(c)** Measurements of individual proteins, IFN-γ and IL-2, reveal no distinguishable cross-reactivity. **(d)** Comparison of the microfluidics flow-patterned DEAL microarrays with DEAL microarrays patterned using a conventional DNA pin-spotting method. The spot size is ~150-200 µm. **(e)** Fluorescence line profiles for the DEAL barcode array in **a** and the pin-spotted array in **d** at different protein concentrations. The curves were amplified in the y-coordinates for better visualization.

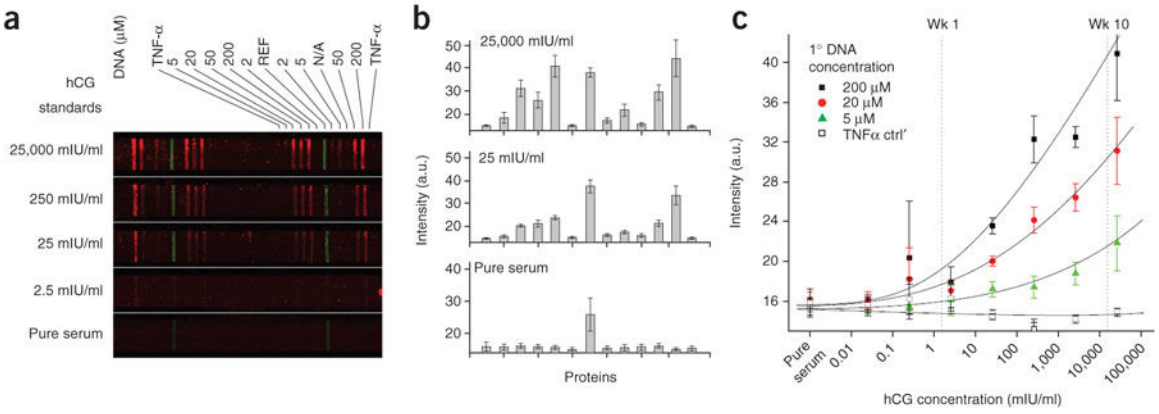


Figure 2.5.6 (a) Fluorescence images of DEAL barcodes showing the measurement of a series of standard serum samples spiked with hCG. The bars used to measure hCG were patterned with DNA strand A at different concentrations. TNF-α encoded by strand B was employed as a negative control. The green bars (strand M) serve as references. **(b)** Quantification of the full barcodes for three selected samples. **(c)** Mean values of fluorescence signals corresponding to three sets of bars with different DNA loadings. Broken lines indicate the typical physiological levels of hCG in sera after 1 or 10 weeks of pregnancy. Error bars, 1 s.d.

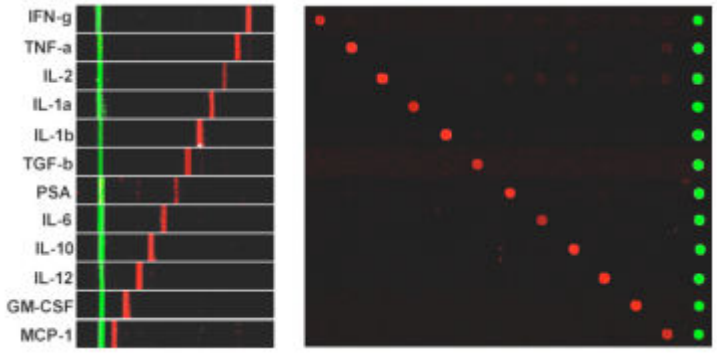


Figure 2.5.7 Cross-reactivity check for all 12 proteins, for both barcode (left panel) and pin-spotted (right panel) microarrays. The green bars represent the reference stripe/spot – M. Each protein can be readily identified by its distance from the reference.

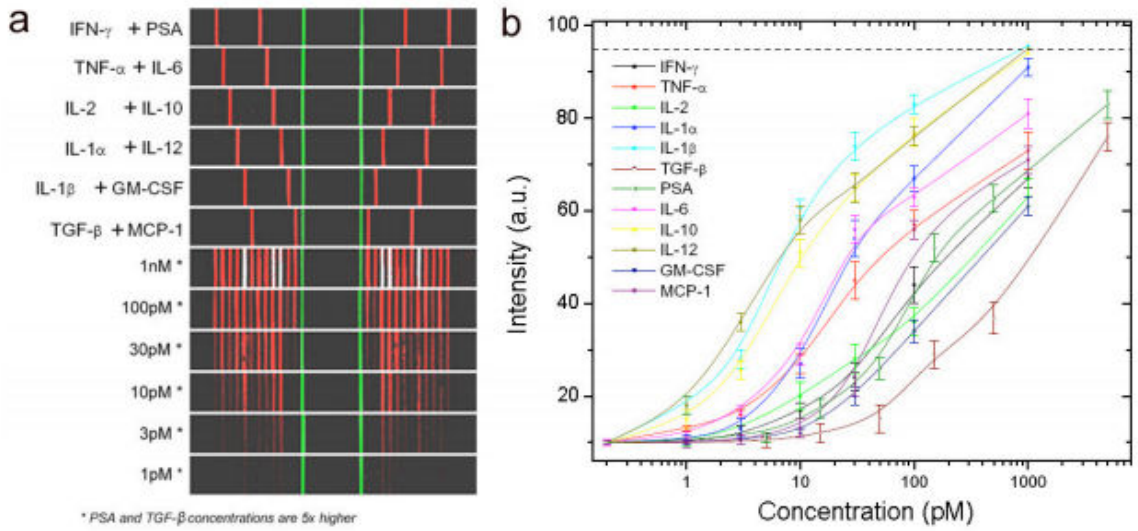


Figure 2.5.8 Dilution curves for 12 proteins measured using DEAL-based barcodes entrained within microfluidics channels. **(a)** Barcode images from one device showing minimal cross-talk, and a series of standard antigens ranging from 1nM to 1pM for all 12 proteins (* the concentrations of PSA and TGF-β are 5x higher). **(b)** Quantitation of fluorescence intensity vs. concentration for all 12 proteins. Error bars: 1SD

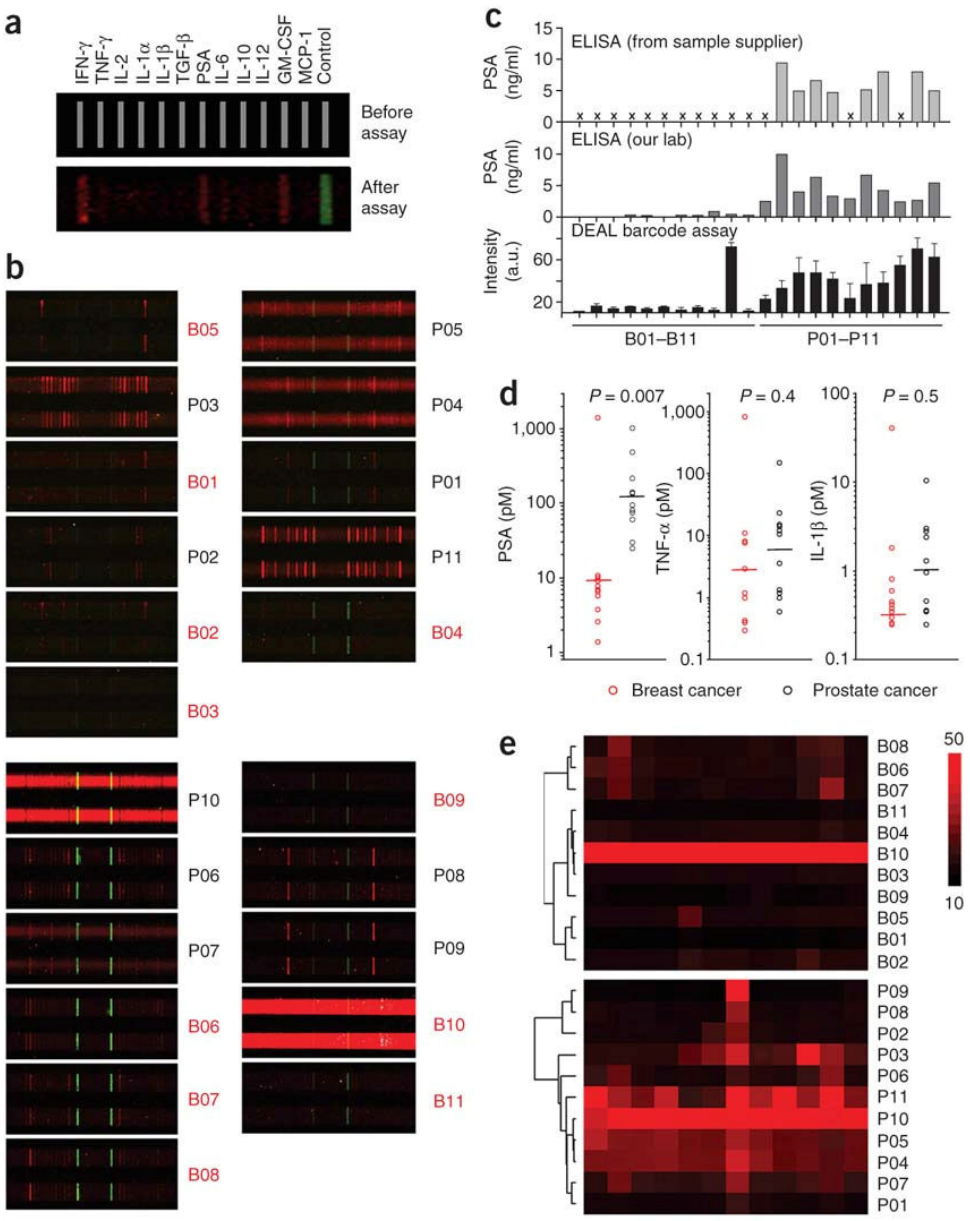


Figure 2.5.9 (a) Layout of the barcode array used in this study. Green denotes the reference (strand M). **(b)** Representative fluorescence images of barcodes used to measure the cancer marker PSA and 11 cytokines from 22 cancer patient serum samples. B01–B11, samples from breast cancer patients; P01–P11, samples from prostate cancer patients. The left and right columns represent measurements on different chips. **(c)** Validation of PSA DEAL barcode measurement using ELISA. x denotes PSA measurements were not provided by the serum supplier. Error bars, 1 s.d. **(d)** Distribution of estimated concentrations of PSA, TNF-α and IL-1β in all serum samples. The horizontal bars mark the mean values. **(e)** Complete nonsupervised clustering of breast and prostate cancer patients on the basis of protein patterns.

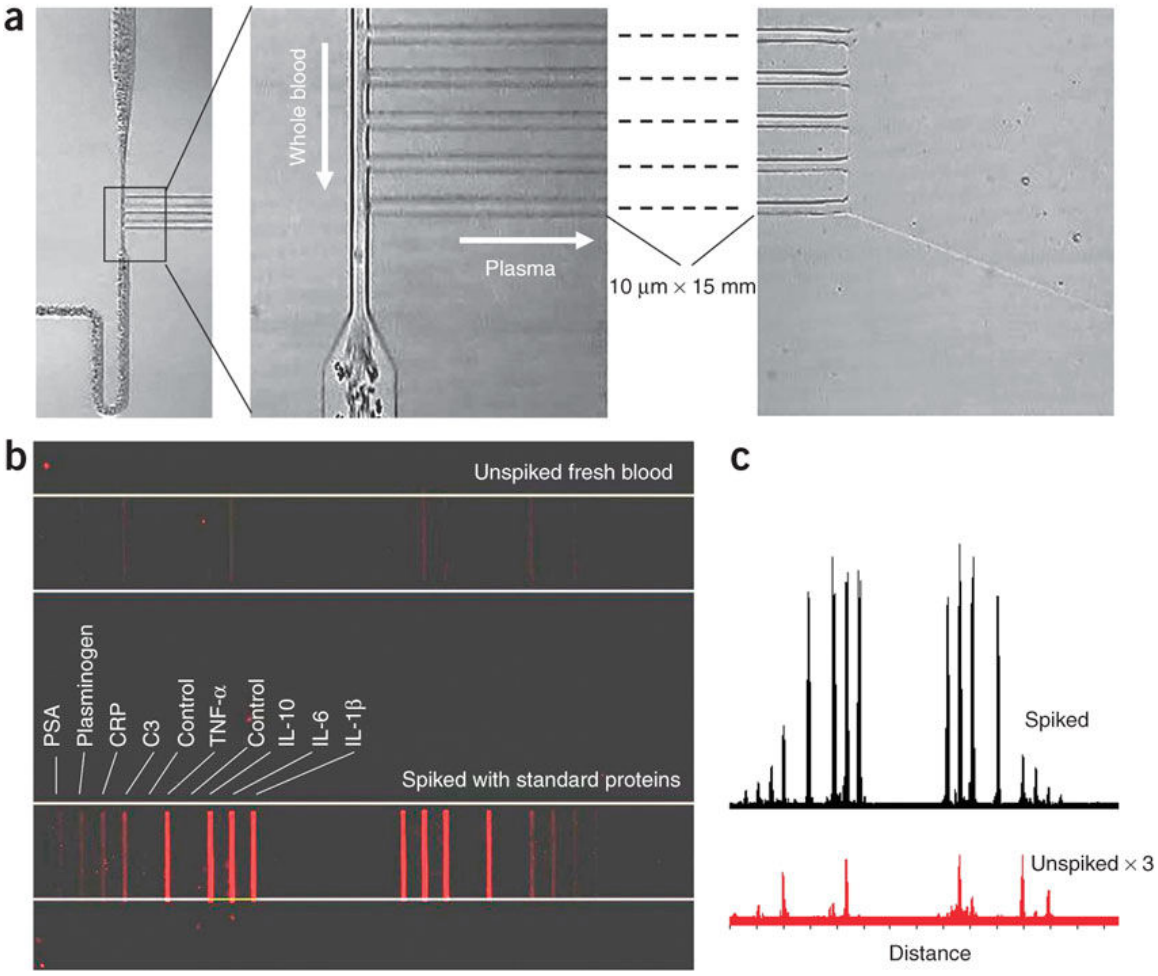


Figure 2.5.10 (a) Optical micrographs showing the effective separation of plasma from fresh whole blood. A few red blood cells occasionally seen downstream of the plasma channels did not affect the protein assay. **(b)** Fluorescence image of blood barcodes in two adjacent microchannels of an IBBC, on which both the unspiked and spiked fresh whole blood collected from a healthy volunteer were separately assayed. Eight plasma proteins are indicated. All bars, 20 µm wide. **(c)** Fluorescence line profiles of the barcodes for both unspiked and spiked whole blood samples. The distance corresponds to the full length shown in b.

2.6 Tables

DNA Code	Human Plasma Protein	Abbreviation
Panel 1		
A/A'	Interferon-gamma	IFN-γ
B/B'	Tumor necrosis factor-alpha	TNF-α
C/C'	Interleukin-2	IL-2
D/D'	Interleukin-1 alpha	IL-1α
E/E'	Interleukin-1 beta	IL-1β
F/F'	Transforming growth factor beta	TGF-β
G/G'	Prostate specific antigen (total)	PSA
H/H'	Interleukin-6	IL-6
I/I'	Interleukin-10	IL-10
J/J'	Interleukin-12	IL-12
K/K'	Granulocyte-macrophage colony stimulating factor	GMCSF
L/L'	Monocyte chemoattractant protein -1	MCP-1
M/M'	Blank control/reference	
Panel 2		
AA/AA'	Interleukin-1 beta	IL-1β
BB/BB'	Interleukin-6	IL-6
CC/CC'	Interleukin-10	IL-10
DD/DD'	Tumor necrosis factor-alpha	TNF-α
EE/EE'	Complement Component 3	C3
FF/FF'	C-reactive protein	CRP
GG/GG'	Plasminogen	Plasminogen
HH/HH'	Prostate specific antigen (total)	PSA

Table 2.6.1 List of protein panels and corresponding DNA codes.

Name	Sequence	Tm °C (50mM NaCl)
A	5'- AAA AAA AAA AAA AAT CCT GGA GCT AAG TCC GTA-3'	57.9
A'	5' NH3- AAA AAA AAA ATA CGG ACT TAG CTC CAG GAT-3'	57.2
B	5'-AAA AAA AAA AAA AGC CTC ATT GAA TCA TGC CTA -3'	57.4
B'	5' NH3AAA AAA AAA ATA GGC ATG ATT CAA TGA GGC -3'	55.9
C	5'- AAA AAA AAA AAA AGC ACT CGT CTA CTA TCG CTA -3'	57.6
C'	5' NH3-AAA AAA AAA ATA GCG ATA GTA GAC GAG TGC -3'	56.2
D	5'-AAA AAA AAA AAA AAT GGT CGA GAT GTC AGA GTA -3'	56.5
D'	5' NH3-AAA AAA AAA ATA CTC TGA CAT CTC GAC CAT -3'	55.7
E	5'-AAA AAA AAA AAA AAT GTG AAG TGG CAG TAT CTA -3'	55.7
E'	5' NH3-AAA AAA AAA ATA GAT ACT GCC ACT TCA CAT -3'	54.7
F	5'-AAA AAA AAA AAA AAT CAG GTA AGG TTC ACG GTA -3'	56.9
F'	5' NH3-AAA AAA AAA ATA CCG TGA ACC TTA CCT GAT -3'	56.1
G	5'-AAA AAA AAA AGA GTA GCC TTC CCG AGC ATT-3'	59.3
G'	5' NH3-AAA AAA AAA AAA TGC TCG GGA AGG CTA CTC-3'	58.6
H	5'-AAA AAA AAA AAT TGA CCA AAC TGC GGT GCG-3'	59.9
H'	5' NH3-AAA AAA AAA ACG CAC CGC AGT TTG GTC AAT-3'	60.8
I	5'-AAA AAA AAA ATG CCC TAT TGT TGC GTC GGA-3'	60.1
I'	5' NH3-AAA AAA AAA ATC CGA CGC AAC AAT AGG GCA-3'	60.1
J	5'-AAA AAA AAA ATC TTC TAG TTG TCG AGC AGG-3'	56.5
J'	5' NH3-AAA AAA AAA ACC TGC TCG ACA ACT AGA AGA-3'	57.5
K	5'-AAA AAA AAA ATA ATC TAA TTC TGG TCG CGG-3'	55.4
K'	5' NH3-AAA AAA AAA ACC GCG ACC AGA ATT AGA TTA-3'	56.3
L'	5' NH3-AAA AAA AAA AGC CGA AGC AGA CTT AAT CAC-3'	57.2
M	5'-Cy3-AAA AAA AAA AGT CGA GGA TTC TGA ACC TGT-3'	57.6
M'	5' NH3-AAA AAA AAA AAC AGG TTC AGA ATC CTC GAC-3'	56.9
AA'	5' NH3-AAAAAAAAAAGTCACAGACTAGCCACGAAG-3'	58
BB	5'-AAA AAA AAA AGC GTG TGT GGA CTC TCT CTA-3'	58.7
BB'	5' NH3-AAA AAA AAA ATA GAG AGA GTC CAC ACA CGC-3'	57.9
CC	5'-AAA AAA AAA ATC TTC TAG TTG TCG AGC AGG-3'	56.5
CC'	5' NH3-AAA AAA AAA ACC TGC TCG ACA ACT AGA AGA-3'	57.5
DD	5'-AAA AAA AAA AGA TCG TAT GGT CCG CTC TCA-3'	58.8
DD'	5' NH3-AAA AAA AAA ATG AGA GCG GAC CAT ACG ATC-3'	58

Table 2.6.2 List of DNA sequences used for spatial encoding of antibodies (continued on next page).

Name	Sequence	Tm °C (50mM NaCl)
EE	5'-AAA AAA AAA AGC ACT AAC TGG TCT GGG TCA-3'	59.2
EE'	5' NH3-AAA AAA AAA ATG ACC CAG ACC AGT TAG TGC-3'	58.4
FF	5'-AAA AAA AAA ATG CCC TAT TGT TGC GTC GGA-3'	60.1
FF'	5' NH3-AAA AAA AAA ATC CGA CGC AAC AAT AGG GCA-3'	60.1
GG	5'-AAA AAA AAA ACT CTG TGA ACT GTC ATC GGT-3'	57.8
GG'	5' NH3-AAA AAA AAA AAC CGA TGA CAG TTC ACA GAG-3'	57
HH	5'-AAA AAA AAA AGA GTA GCC TTC CCG AGC ATT-3'	59.3
HH'	5' NH3-AAA AAA AAA AAA TGC TCG GGA AGG CTA CTC-3'	58.6

Table 2.6.2 List of DNA sequences used for spatial encoding of antibodies.

PATIENT	CANCER	GENDER/AGE	RACE	UICC STAGE	GLEASONS SCORE	OTHERS
B01	Breast	Female/62	Caucasian	T2N0M0		wine 200mL/day
B02	Breast	Female/79	Caucasian	T4N2M0		
B03	Breast	Female/71	Caucasian	T1cNXM0		1-2 drinks/day
B04	Breast	Female/72	Caucasian	T2NXM0		hypertension
B05	Breast	Female/89	Caucasian	T3N0MX		arthritis
B06	Breast	Female/56	Asian	T1NXM0		
B07	Breast	Female/54	Caucasian	T2N2M0		hypertension, obesity
B08	Breast	Female/55	Caucasian	T2NxM0		1-5 cigs/day, wine 200mL/day
B09	Breast	Female/83	Caucasian	T4N0M0		Coronary artery disease, cerebral atherosclerosis
B10	Breast	Female/63	Hispanic	T3N2MX		6-10cigs/day, hyperthyroid, hypertension, osteoarthritis
B11	Breast	Female/63	Caucasian	T1NXM0		arterial hypertension
P01	Prostate	Male / 51	Caucasian	T2cNXM0	4+3=7	
P02	Prostate	Male / 64	Caucasian	T3bN0MX	3+4=7	
P03	Prostate	Male / 47	Caucasian	T2cN0M0	3+3=6	hypertension
P04	Prostate	Male / 55	Caucasian	T2bN0M0	3+3=6	11-20 cigs/day
P05	Prostate	Male / 73	Caucasian	T3aNXMX	4+4=8	hypertension,11-20 cigs/day
P06	Prostate	Male/64	Caucasian	T3N0M0		chronic bronchitis, 11-20cigs/day
P07	Prostate	Male/60	Caucasian	T3aN0M0	3+4=7	gastroesophageal reflux
P08	Prostate	Male/72	African Am.	T2aNXMX	3+3=6	1-5cigs/day
P09	Prostate	Male/78	Caucasian	T3aN1MX	4+3=7	hypertension, atrial fibrillation
P10	Prostate	Male/66	Caucasian	T2aN0MX	3+3=6	hypertension, 11-20 cigs/day
P11	Prostate	Male / 47	Caucasian	T2cN0M0	3+3=6	hypertension

Table 2.6.3 Medical records of cancer patients.

2.7 References

1 Anderson, N. L. & Anderson, N. G. The human plasma proteome: history, character, and diagnostic prospects. *Mol. Cell. Proteomics* **1**, 845-867, (2002).

2 Fujii, K. Clinical-scale high-throughput human plasma proteome clinical analysis: Lung adenocarcinoma. *Proteomics* **5**, 1150-1159, (2005).

3 Lathrop, J. T., Anderson, N. L., Anderson, N. G. & Hammond, D. J. Therapeutic potential of the plasma proteome. *Curr. Opin. Mol. Ther.* **5**, 250-257, (2003).

4 Chen, J. H. Plasma proteome of severe acute respiratory syndrome analyzed by two-dimensional gel electrophoresis and mass spectrometry. *Proc. Natl. Acad. Sci. USA* **101**, 17039-17044, (2004).

5 Hsieh, S. Y., Chen, R. K., Pan, Y. H. & Lee, H. L. Systematical evaluation of the effects of sample collection procedures on low-molecular-weight serum/plasma proteome profiling. *Proteomics* **6**, 3189-3198, (2006).

6 Sia, S. K. & Whitesides, G. M. Microfluidic devices fabricated in poly(dimethylsiloxane) for biological studies. *Electrophoresis* **24**, 3563-3576, (2003).

7 Quake, S. R. & Scherer, A. From micro- to nanofabrication with soft materials. *Science* **290**, 1536-1540, (2000).

8 Huang, B. Counting low-copy number proteins in a single cell. *Science* **315**, 81-84, (2007).

9 Ottesen, E. A., Hong, J. W., Quake, S. R. & Leadbetter, J. R. Microfluidic digital PCR enables multigene analysis of individual environmental bacteria. *Science* **314**, 1464-1467, (2006).

10 Huang, L. R., Cox, E. C., Austin, R. H. & Sturm, J. C. Continuous particle separation through deterministic lateral displacement. *Science* **304**, 987-990, (2004).

11 Chou, C. F. Sorting biomolecules with microdevices. *Electrophoresis* **21**, 81-90, (2000).

12 Toner, M. & Irimia, D. Blood-on-a-chip. *Annu. Rev. Biomed. Eng.* **7**, 77-103, (2005).

13 Nagrath, S. Isolation of rare circulating tumour cells in cancer patients by microchip technology. *Nature* **450**, 1235-1239, (2007).

14 Yang, S., Undar, A. & Zahn, J. D. A microfluidic device for continuous, real time blood plasma separation. *Lab Chip* **6**, 871-880, (2006).

15 Svanes, K. & Zweifach, B. W. Variations in small blood vessel hematocrits produced in hypothermic rates by micro-occlusion. *Microvasc. Res.* **1**, 210-220, (1968).

16 Fung, Y. C. Stochastic flow in capilary blood vessels. *Microvasc. Res.* **5**, 34-38, (1973).

17 Bailey, R. C., Kwong, G. A., Radu, C. G., Witte, O. N. & Heath, J. R. DNA-encoded antibody libraries: a unified platform for multiplexed cell sorting and detection of genes and proteins. *J. Am. Chem. Soc.* **129**, 1959-1967, (2007).

18 Boozer, C., Ladd, J., Chen, S. F. & Jiang, S. T. DNA-directed protein immobilization for simultaneous detection of multiple analytes by surface plasmon resonance biosensor. *Anal. Chem.* **78**, 1515-1519, (2006).

19 Niemeyer, C. M. Functional devices from DNA and proteins. *Nano Today* **2**, 42-52, (2007).

20 Pirrung, M. C. How to make a DNA chip. *Angewandte Chemie-International Edition* **41**, 1276-1289, (2002).

21 Coussens, L. M. & Werb, Z. Inflammation and cancer. *Nature* **420**, 860-867, (2002).

22 Lin, W. W. & Karin, M. A cytokine-mediated link between innate immunity, inflammation, and cancer. *J. Clin. Invest.* **117**, 1175-1183, (2007).

23 De Marzo, A. M. Inflammation in prostate carcinogenesis. *Nat. Rev. Cancer* **7**, 256-269, (2007).

24 Ashton, H. & Telford, R. Smoking and carboxhemoglobin. *Lancet* **2**, 857-858, (1973).

25 Chopra, V., Dinh, T. V. & Hannigan, E. V. Serum levels of interleukins, growth factors and angiogenin in patients with endometrial cancer. *J. Cancer Res. Clin. Oncol.* **123**, 167-172, (1997).

26 Oncul, O., Top, C. & Cavuplu, P. Correlation of serum leptin levels with insulin sensitivity in patients with chronic hepatitis-C infection. *Diabetes Care* **25**, 937, (2002).

27 Pinsky, M. R. Serum cytokine levels in human septic shock: relation to multiple-system organ failure and mortality. *Chest* **103**, 565-575, (1993).

28 Schweitzer, B. Multiplexed protein profiling on microarrays by rolling-circle amplification. *Nat. Biotechnol.* **20**, 359-365, (2002).

29 Lambeck, A. J. A. Serum cytokine profiling as a diagnostic and prognostic tool in ovarian cancer: A potential role for interleukin 7. *Clin. Cancer Res.* **13**, 2385-2391, (2007).

30 Gorelik, E. Multiplexed immunobead-based cytokine profiling for early detection of ovarian cancer. *Cancer Epidemiol. Biomarkers Prev.* **14**, 981-987, (2005).

31 Heath, J. R. & Davis, M. E. Nanotechnology and cancer. *Annu. Rev. Med.* **59**, 251-265, (2008).

32 Zimmermann, M., Delamarche, E., Wolf, M. & Hunziker, P. Modeling and optimization of high-sensitivity, low-volume microfluidic-based surface immunoassays. *Biomed. Microdevices* **7**, 99-110, (2005).

33 Eisen, M. B., Spellman, P. T., Brown, P. O. & Botstein, D. Cluster analysis and display of genome-wide expression patterns. *Proc. Nat. Acad. Sci. USA* **95**, 14863-14868, (1998).

Chapter 3

Chemistries for Patterning Robust DNA MicroBarcodes Enable Multiplex Assays of Cytoplasm Proteins from Single Cancer Cells

3.1 Introduction

The demand for parallel, multiplex analysis of protein biomarkers from ever smaller biospecimens is an increasing trend for both fundamental biology and clinical diagnostics[1-3]. The most highly multiplex protein assays rely on spatially encoded antibody microarrays[4-6], and small biospecimens samples are now routinely manipulated using microfluidics approaches. The integration of antibody microarray techniques with microfluidics chips has only been explored relatively recently. One challenge arises from the relative instability of antibodies to microfluidics fabrication conditions. In recent years, several groups have devised methods to transform standard DNA microarrays in situ into protein microarrays and cell-capture platforms[7-12]. These approaches capitalize on the chemical robustness of DNA oligomers, and the reliable assembly of DNA-labeled structures via complementary hybridization. Recently, Fan et al. utilized a microfluidics-based flow patterning technique to generate DNA barcode-

type arrays at 10× higher density than standard, spotted microarrays[13]. The DNA barcodes were converted into antibody arrays using the DNA-encoded antibody library (DEAL) technique, and then applied towards the measurement of a highly multiplex panel of proteins from a pinprick of whole blood.

A second challenge involves scaling such miniaturized DNA microarrays so that a large surface area can be encoded. This problem is non-trivial, as it involves identifying chemistries for patterning 10^{-5} m wide, 1 m long strips of biomolecules with a uniformity that permits those patterns to be utilized in hundreds to thousands of quantitative protein assays per chip. Herein, we explore the surface chemistry associated with microfluidics-based flow patterning of DNA barcodes, with an eye towards producing highly reproducible and robust barcodes. We then apply the optimized chemistry towards assaying a panel of cytoplasmic proteins from single cells.

We explore three different flow patterning surface chemistries: two rely upon the electrostatic adsorption of DNA onto a poly-L-lysine (PLL) surface, and the third utilizes flow patterning of dendrimers onto aminated glass substrates, followed by covalent attachment of DNA oligomers onto the dendrimer scaffolds. For the electrostatic adsorption cases, we investigate, using both theory and experiment, the role that counterions play in flow patterning within the confined dimensions of a microfluidic channel, and we find that solvent mixtures which associate counterions more strongly to the negatively charged DNA oligomers yield more reproducible and robust barcodes.

We then demonstrate the utility of the best flow patterning chemistry by combining it with DEAL to construct antibody barcodes for quantitatively assaying a panel of phosphorylated proteins, associated with oncogenic pathways, from single cells that are representative of the brain cancer glioblastoma multiforme (GBM).

3.2 Results and Discussion

3.2.1 Device design and functionalization schemes

The microfluidics flow patterning chip is comprised of a patterned polydimethylsiloxane (PDMS) layer adhered to an aminated or PLL-coated glass substrate that provides the base surface for the microchannels. The microchannels are long (about 55 cm), meandering channels that span ca. 0.85 cm^2 of our substrate, and are used to pattern a DNA barcode over most of the glass surface (Figure 3.5.1b). After the flow patterning is completed, the PDMS layer is replaced with a second micropatterned PDMS layer that is designed to support a biological assay, such as the previously reported blood proteomics chip[13], or the single-cell proteomics chips utilized herein. For the microfluidic patterning method to be useful, it must generate a DNA barcode that exhibits high and uniform DNA loading over the entire substrate. We evaluated the patterning chemistries illustrated in Figure 3.5.1a, Schemes 1–3. Schemes 1 and 2 are drawn from the conventional protocol for pin-spotted microarrays—a solution containing the DNA is introduced, the solvent is evaporated, and subsequent thermal or

UV treatment is employed to cross-link the deposited DNA to the substrate. In Scheme 1 ssDNA oligomers dissolved in phosphate-buffered saline (PBS) are utilized, whereas in Scheme 2 ssDNAs in a 1:1 mixture of 1×PBS and dimethyl sulfoxide (DMSO) are employed. DMSO is used in conventional microarray preparation to improve feature consistency by reducing the rate of solvent evaporation and by denaturing the DNA[14] although, as described below, its role in this process is different. In Scheme 3 a covalent immobilization method based upon a dendrimer scaffold is utilized[15]. Poly(amidoamine) (PAMAM) dendrimers (generation 4.5, carboxylate surface) have previously shown promise as DNA and protein microarray substrates. Dendrimers do not form entangled chains[16] and because harsh crosslinking procedures are avoided, dendrimer-immobilized DNA retains high accessibility and activity in microarray applications. Moreover, the highly branched structure of the dendrimers provides a high density of reactive sites for surface attachment and for DNA coupling, thus leading to a high overall binding capacity. For all cases, a high level of DNA loading has been shown to decrease non-specific binding when compared to standard microarray substrates[10,17-19].

Figure 3.5.1b (top) shows the PDMS chip design used for barcode patterning. Thirteen discrete channels (for a thirteen-element barcode) allow for a multiplex microarray. We loaded five adjacent channels according to Scheme 1, skipped three channels, and then loaded the remaining five channels according to Scheme 2. The use of fluorescently-tagged DNA permitted measurements of the DNA distribution within each individual

channel immediately after introducing the solutions. Figure 3.5.1b demonstrates a clear difference in aqueous DNA distribution across the chip: DNA loaded according to Scheme 1 (outer five channels) is notably lower in concentration near the middle of the chip (Figure 3.5.1b, Region 2) and is barely detectable near the channel exit (Figure 3.5.1b, Region 1). Conversely, DNA loaded according to Scheme 2 (inner five channels) presents an even, consistent distribution across the entire chip. Notably, Scheme 1 yields a relatively higher fluorescence intensity at the input side of the chip. These results clearly indicate that, for Scheme 1, the ssDNA oligomers are accumulating upstream during the early stages of flow, and so are depleted from the advancing solution by the time it reaches mid-chip. The actual patterning of the glass substrate occurs when solvent is evaporated (Figure 3.5.2). Indeed, the final patterning results after solvent evaporation and cross-linking (Figure 3.5.1c, top) reflect the trend established by the aqueous fluorescence images; Scheme 2 produces uniform DNA barcodes across the substrate, while Scheme 1 does not.

3.2.2 DMSO mechanism and simulations

In order to understand the difference in patterning uniformity between Schemes 1 and 2, we considered the electrostatic environment for each case. As depicted in Figure 3.5.3a, the PDMS side walls carry a slightly negative zeta potential, whereas the PLL surface has a strong positive zeta potential[20]. When the ssDNA solution in Scheme 1 is introduced to the channel, ssDNA near the PLL matrix is electrostatically immobilized, thereby generating a concentration gradient[21]. As the solution flows towards the

channel exit, the ssDNA oligomers are continually depleted via deposition onto the PLL surface. Figure 3.5.3b shows the results from a rough simulation designed to capture the mean concentration of aqueous ssDNA as the solution traverses a channel. The simulation implies that the effect of electrostatic adsorption proves dominant even at high DNA concentrations, a result that agrees well with the observed behavior for Scheme 1 in Figure 3.5.1b. A detailed description of the model and assumptions employed can be found in the Supporting Information. We tested this model via the strong negative charging of all four channel surfaces via O_2 plasma treatment. Consistent with the model, both Schemes 1 and 2 exhibited equivalently uniform distribution of fluorescence intensity across the chip (Figure 3.5.4b). We note that lack of the positive charges on the bottom surface failed to hold DNAs during the drying procedure and that the plasma treatment induces the irreversible bonding of PDMS and glass, which limits further use beyond this experimental test.

The results from Schemes 1 and 2 imply that DMSO alleviates the electrostatic adsorption effect. In order to understand this more fully, we performed molecular dynamics (MD) simulations of DNA in PBS and PBS/DMSO solutions; 3 ns of NPT [NPT is a simulation in which number of moles (N), pressure (P) and temperature (T) are held constant]. The MD simulations were performed with the last 1 ns trajectory used for analysis. We examined the radial distribution function of phosphorous atoms in the DNA backbone with respect to various elements of the surrounding solvent. For example, the radial distribution function of P and the O atom of a water molecule is virtually

unperturbed by the addition of DMSO (Figure 3.5.5). Consequently, it is unsurprising that the radial distribution function of P and the S atom of DMSO (Figure 3.5.3c, black solid line) reveals that DMSO is not forming a solvation structure with the DNA backbone. However, Figure 3.5.2c demonstrates a clear interaction between P and Na^+ ions, which delineates into two well-defined shell structures: the first is located at r<4.3 Å while the second is located at 4.3 Å<r<6.6 Å. These are similar to the locations of the first and the second water solvation structures. By integrating the radial distribution functions, we determined the number of molecules per phosphate in the first and second shells for both PBS and PBS/DMSO solutions. Although the number H_2O molecules per shell is virtually independent of DMSO, DMSO does significantly increase the number of Na^+ ions in the first shell (from 0.14 to 0.24), and it decreases the number of Na^+ ions in the second shell (from 0.61 to 0.34). Conversely, the number of DMSO molecules is almost zero in the first shell (0.01) but becomes significant in the second shell (0.20). Thus, we conclude that DMSO, with a lower dielectric constant relative to water (47.2 vs 80), destabilizes the solvation energy of Na^+ in the second shell. This thermodynamic change prompts the sodium ions to move to the first shell where they are stabilized by electrostatic interactions with the negatively charged phosphate groups. The increased number of sodium ions near the DNA backbone screens the negative charges of phosphate groups more efficiently, thereby reducing electrostatic interactions of the DNA with the PLL surface, resulting in uniform DNA distribution throughout the channels. Although the addition of DMSO to DNA patterning solutions yields the same ultimate effect for both traditional spotted arrays and microfluidics-

patterned barcodes, the underlying mechanisms are completely different. We conclude that Scheme 2 is intrinsically superior relative to Scheme 1.

3.2.3 Covalent attachment mechanism and comparision

We now turn towards analyzing Scheme 3, and comparing it against Scheme 2. For this scheme, the PAMAM dendrimers are first covalently attached to the aminated glass surface, and then (aminated) ssDNA oligomers are covalently attached to the dendrimers. The lack of a solvent evaporation step makes Scheme 3 significantly more rapid than Scheme 2. We flowed activated PAMAM dendrimers, followed by aminated ssDNA, through ten microfluidic channels (Figure 3.5.1b). Note that the aqueous DNA distribution is expected to be uniform because the substrate surface is comprised of charge-neutral N-hydroxysuccinimide (NHS)-modified carboxylates which minimize electrostatic interactions. The resulting DNA microarray was assayed for uniformity with complementary DNAs labeled with Cy3-fluorophores. Visual analysis indicates good uniformity across the chip (Figure 3.5.1c, bottom). In order to quantify the patterning quality for all three schemes, we obtained signal intensities for each channel at sixteen locations within the patterning region and calculated the coefficient of variation (CV). The CV is defined as the standard deviation divided by the mean and expressed as a percentage. CVs for Schemes 1, 2, and 3 registered 69.8%, 10.5%, and 10.9%, respectively. Thus, we conclude that Schemes 2 and 3 offer consistent DNA loading across the entire substrate.

Having established that Schemes 2 and 3 produce consistent, large-scale DNA barcodes, we then extended our analysis of array consistency to protein measurements. We previously demonstrated that, when using the DEAL platform for multiplex protein sensing in microfluidics channels, the sensitivities of the assays directly correlate with the amount of immobilized DNA[13], up to the point where the DNA coverage is saturated. We performed multiple protein assays along the length of our DNA stripes to ensure that the results described above would translate into stable and sensitive barcodes for protein sensing. All protein assays were performed in microfluidic channels which were oriented perpendicular to the patterned barcodes (five channels for Scheme 2 and four channels for Scheme 3). This allowed us to test distal microarray repeats with a single small analyte volume. For barcodes prepared using Scheme 2, we utilized the DEAL technique to convert them into antibody barcodes designed to assay the following proteins: phosphorylated (phospho)-steroid receptor coactivator (Src), phospho-mammalian target of rapamycin (mTOR), phospho-p70 S6 kinase (S6K), phospho-glycogen synthase kinase (GSK)-3α/β, phospho-p38α, phospho-extracellular signal-regulated kinase (ERK), and total epidermal growth factor receptor (EGFR) at 10 ng/mL^{-1} and 1 ng/mL^{-1} concentrations. This panel samples key nodes of the phosphoinositide 3-kinase (PI3K) signaling pathway within GBM, and are used below for single-cell assays[22]. For barcodes prepared using Scheme 3, we similarly converted the DNA barcodes into antibody barcodes designed to detect three proteins [interferon (INF)-γ, tumor necrosis factor (TNF)α, and interleukin (IL)-2] at 100 ng/mL^{-1} and 10 ng/mL^{-1}. All the DNAs used were pre-validated for the orthogonality in order to avoid

cross-hybridization and the sequences can be found in Table 3.6.1. The detection scheme is similar to a sandwich immunoassay. Captured proteins from primary antibodies were fluorescently visualized by biotin-labeled secondary antibodies and Cy5-labeled streptavidin. For both cases, data averaged from multiple DNA repeats across the chip yielded CVs that were commensurate with those of the underlying DNA barcodes (from 10 ng/mL^{-1} concentration, 7% for scheme 2 and 17% for Scheme 3, respectively). Figure 3.5.6 shows line profiles of the signal intensities along with the raw data, and demonstrate a better uniformity for barcodes prepared according to Scheme 2. While we found that Scheme 3 could produce barcodes that were close in quality to those of Scheme 2, the absolute (chip-to-chip) consistency of Scheme 3 is hard to guarantee due to its use of the unstable coupling reagents 1-ethyl-3-(3-dimethylaminopropyl) carbodiimide (EDC) and NHS[23]. Moreover, although Scheme 3 is faster, the detailed procedure itself is more labor-intensive. Scheme 2 can potentially be automated. Thus, we chose Scheme 2 as the preferred barcode patterning method. With Scheme 2 , over 90% of the patterned slides showed good quality for the test.

3.2.4 Single cell assays

We validated the use of the antibody barcodes by applying them towards the multiplex assay of cytoplasmic proteins from single cells. There is a significant body of evidence that demonstrates that genetically identical cells can exhibit significant functional

heterogeneity—behavior that cannot be captured by proteomics techniques that average data across a population[24]. We therefore designed a highly parallel microfluidic device capable of isolating single/few numbers of cells in chambers with a full complement of antibody barcodes designed to detect intracellular proteins (Figure S5, Supporting Information). Figure 3.5.7a shows a schematic of the device and the DEAL-based protein detection scheme. The small chamber size keeps the finite number of protein molecules concentrated, thereby enhancing sensitivity. Assaying such a panel of proteins would not be possible without a high density antibody array, such as the barcodes utilized herein, for the following reasons. First, all the barcodes should fit into such a small chamber for multiplexing. Second, since data averaging in such a spatially-constrained scheme is impractical, it is critical to have consistent DNA loading across the microrarray if data comparisons are to be meaningful.

We chose the U87 GBM cell line as a model system for our platform. GBM is the most common malignant brain tumor found in adults, and is the most lethal of all cancers. As the name implies, GBM exhibits extensive biological variability and heterogeneous clinical behavior[25]. EGFR is an important GBM oncogene and therapeutic target[26]. Thus, we assayed for eleven intracellular proteins associated with the EGFR-activated PI3K signaling pathway. We provide representative sets of data for protein detection from the lysate of one to five cells (Figures 3.5.7b and c). Eight proteins were detected from single-cell lysate and up to nine proteins were detected from five cells when using barcodes patterned by Scheme 2 (Figures 3.5.7b,d), whereas only one protein could be

detected from barcodes prepared by Scheme 1 (Figure 3.5.7c). All the separate protein assays were screened for cross-reactivity (Figure 3.5.8), and, for the cases where recombinant proteins were available, quantitation curves for each protein assay were measured (Figure 3.5.9). More detailed statistical analysis of these cells, as well as genetic variants thereof, is currently being investigated.

3.3 Conclusions

We identified a protocol for generating high-quality, high-density DNA barcode patterns by comparing three microfluidics-based patterning schemes. We find, through both experiment and theory, that the electrostatic attractions between positively-charged PLL substrates and the negatively-charged DNA backbone induces significant non-uniformity in the patterning process, but that those electrostatic interactions may be mediated by adding DMSO to the solution, resulting in uniform and highly reproducible barcodes patterned using ~55 cm long channels that template barcodes across an entire 2.5 cm wide glass slide. Dendrimer-based covalent immobilization also yields good ultimate uniformity, but is hampered by a relatively unstable chemistry that limits run-to-run reproducibility. DNA barcodes were coupled with the DEAL technique to generate antibody barcodes, and then integrated into specifically designed microfluidic chips for assaying cytoplasm proteins from single and few lysed U87 model cancer cells.

Successful detection of a panel of such proteins represents the potential of our platform to be applied to various biological and, perhaps, clinical applications.

3.4 Experimental Section

3.4.1 Microfluidic chip fabrication for DNA patterning.

Microfluidic-patterning PDMS chips were fabricated by soft lithography. The master mold was prepared using either a negative photoresist, SU8 2010, with photolithography or an etched silicon mold generated by a deep reactive ion etching (DRIE) process. The mold has long meandering channels with a 20×20 μm cross section. The distance from channel to channel is also 20 μm, which generates 10× higher density than standard, spotted microarrays. Sylgard PDMS (Corning) prepolymer and curing agent were mixed in a 10:1 ratio (w/w), poured onto the mold, and cured (80°, 1 hour). The cured PDMS slab was released from the mold, inlet/outlet holes were punched, and the device was bonded onto a PLL coated (C40–5257M20, Thermo scientific) or aminated glass slide (48382–220, VWR) to form enclosed channels. The number of microfluidic channels determines the size of the microarray; 13 parallel microchannels were used in this study.

3.4.2 Patterning of DNA barcode arrays.

For the DNA filling test, a 30-mer DNA oligomer labeled with Cy3 fluorescence tag on the 5' end (5'-/Cy3/-AAA AAA AAA ATA CGG ACT TAG CTC CAG GAT-3') in a 1:1 mixture (v/v) of 1×PBS buffer and DMSO or a 1:1 mixture (v/v) of 1×PBS buffer and deionized (DI) water was used. The final DNA concentration was 2.5 μM. DNA solution was pushed into the channel under a constant pressure (2.5 psi). Immediately after the channels were fully filled, fluorescence images were obtained by confocal microscopy.

Dendrimer-based microarrays were prepared using aminated substrates. Generation 4.5 Poly(amidoamine) (PAMAM) dendrimers (470457–2.5G, Aldrich), 5% wt in MeOH, were mixed 1:1 (v/v) with EDC/NHS (0.2M) in MES buffer (0.1M, pH 6.0). After 5 min of incubation, the activated dendrimers were introduced to the microfluidic channels, and allowed to flow (2 h). Following a brief MeOH rinse to remove unbound dendrimers, the channels were filled with EDC/NHS (0.2M) in MES (0.1M, pH 5.3) with NaCl (0.5M). After 0.5 h, 5' aminated DNA sequences in 1×PBS (200 μM) were introduced to the channels and allowed to flow (2 h). Thereafter, the microfluidic device was removed from the substrate, and the latter was rinsed copiously with DI water. Prepared substrates that were not used immediately were stored in a desiccator.

To generate the DNA barcode array for multi-protein detection and single-cell lysis test, 13 orthogonal DNA oligomer solutions (sequences are provided in Table 3.6.1) in 1×PBS buffer (400 μM) were mixed with DMSO (in 1:2 ratio, v/v) and flowed into each of the

microfluidic channels (Scheme 2). For Scheme 1 , DNA solutions in 1×PBS buffer were used. The DNA-filled chip was placed in a desiccator until the solvent evaporated completely, leaving only DNA molecules behind. Finally, the PDMS elastomer was removed from the glass substrate and the microarray-patterned DNAs were cross-linked to the PLL by thermal treatment (80°C, 4 h). The slide was gently rinsed with DI water prior to use in order to remove salt crystals remaining from the solution evaporation step.

3.4.3 Microfluidic chip fabrication for multi-protein detection.

The PDMS microfluidic chip for the cell experiment was fabricated by two-layer soft lithography[27]. A push-down valve configuration was utilized with a thick control layer bonded together with a thin flow layer. The molds for the control layer and the flow layer were fabricated with SU8 2010 negative photoresist (~20 µm thickness) and SPR 220 positive photoresist (~18 µm), respectively. The photoresist patterns for the flow layer were rounded via thermal treatment. The thick control layer was molded with a 5:1 mixture of GE RTV 615 PDMS prepolymer part A and part B (w/w) and the flow layer was formed by spin-coating a 20:1 mixture of GE RTV 615 part A and part B (w/w) on the flow layer mold (2000 rpm, 60 sec). Both layers were cured (80°C, 1 hour), whereupon the control layer was cut from its mold and aligned to the flow layer. An additional thermal treatment (80°C, 1 hour) ensured that the two layers bonded into a monolithic device, which was then peeled from its mold and punched to create appropriate access

holes. Finally, the PDMS chip was thermally bonded to the DNA microbarcodes-patterned glass slide to form the working device.

3.4.4 Cell culture.

The human GBM cell line U87 was cultured in DMEM (American Type Culture Collection, ATCC) supplemented with 10% fetal bovine serum (FBS, Sigma–Aldrich). U87 cells were serum-starved for 1 day and then stimulated by EGF (50 ng/mL−1, 10 min) before they were introduced into the device.

3.4.5 Multi-protein detection.

Protein detection assays were initiated by blocking the chip with 3% bovine serum albumin (BSA) in PBS to prevent non-specific binding. This 3% BSA/PBS solution was used as a working buffer for most subsequent steps. After blocking, a cocktail containing all eleven (Scheme 2) or three (Scheme 3) DNA–antibody conjugates (~0.5 μg/mL^{-1}, 100 μL) in working buffer was flowed through the micro channels for 1 h. The unbound DNA–antibody conjugates were washed away with fresh buffer. Then, target proteins were flowed through the microfluidic channels for 1 hour. These were followed by a 200 μL cocktail containing biotin-labeled detection antibodies (~0.5 μg/mL^{-1}) in working buffer, and thereafter a 200 μL mixture of 1 μg/mL^{-1} Cy5-labeled streptavidin and 25 nMCy3-labeled M’ ssDNA in working buffer to complete the immune sandwich assay. DNA sequence M is used for a location reference. The microchannels were rinsed with working buffer once more before the PDMS chip was removed; the bare microarray

slide was rinsed sequentially with 1×PBS, 0.5×PBS, DI water, and was finally subjected to spin-drying.

3.4.6 On-chip cell lysis and multiplexed intracellular protein profiling from single cells.

The multi-protein detection procedure described above was slightly modified for intracellular protein profiling experiments. Again, the chip was initially blocked with a 3% BSA/PBS working buffer, followed by a 200 μL cocktail containing all eleven DNA–antibody conjugates (~0.5 μg/mL^{-1}, Table 3.6.2) in working buffer (continuously flowed for 1 h). Unbound DNA-antibody conjugates were washed off with fresh buffer. The lysis buffer (Cell Signaling) was loaded into the lysis buffer channels while valve 1 (V1 in Figure 3.5.7a) was kept closed by applying 15–20 psi constant pressure. Then, cells were introduced to the cell loading channels and microfluidic valves (V2 in Figure 3.5.7a) were closed by applying 15–20 psi constant pressure; this converts the eight channels into 120 isolated microchamber sets. After cell numbers were counted under microscope, V1 valves were released to allow diffusion of lysis buffer to the neighboring microchamber containing different numbers of cells. The cell lysis was performed on ice for two hours. After that, the V2 valves were released and the unbound cell lysate was quickly removed by flowing the fresh buffer. Then, a cocktail containing biotin-labeled detection antibodies (~0.5 μg/mL^{-1}, 200 μL) in working buffer was flowed into the chip for 1 h on ice, followed by flowing a 200 μL mixture of Cy5-labeled straptavidin (1 μg/mL^{-1}) and Cy3-labeled M' ssDNA (25 nM) in working buffer to complete the sandwich immunoassay. Finally, the microchannels were rinsed with working buffer, the PDMS

chip was removed, and the bare microarray slide was rinsed sequentially with 1×PBS, 0.5×PBS, DI water, before spin-drying. The layout of the chip and used inlets for different solutions were described in Figure S5.

3.4.7 Data analysis.

The microarray slide was scanned with the GenePix 200B (Axon Instruments) to obtain a fluorescence image of both Cy3 and Cy5 channels. All scans were performed with the same setting of 50% (635 nm) and 15% (532 nm) laser power, 500 (635 nm) and 450 (532 nm) optical gain. The averaged fluorescence intensities for all barcodes in each chamber were obtained and matched to the cell number by custom-developed Excel or MATLAB codes.

3.4.8 Molecular dynamic simulations.

The MD simulations were performed with the all-atom AMBER2003 force field[28,29] using the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) code[30]. As an initial structure, a single strand of DNA (5'-ACCCATGGAGCATTCCGGG-3') whose base pairs were randomly chosen was built using Namot2 program[31]. Near the DNA strand, 19 sodium counter ions were included to neutralize the negatively charged 19 phosphate groups on the DNA backbone. Then, this is immersed in a solvation box composed of either 1) 5206 water molecules+106 DMSO molecules or 2) only 5206 water molecules. We used TIP4P model to describe the water interactions[32]. We

performed 3 ns NPT MD simulations using Nosé–Hoover thermostat with a damping relaxation time of 0.1 ps and Andersen–Hoover barostat with a dimensionless cell mass factor of 1.0. The last 1 ns trajectory is employed for the analysis. To compute the electrostatic interactions, the particle-particle particle-mesh method[33] was employed using an accuracy criterion of 10^{-4}.

3.5 Figures

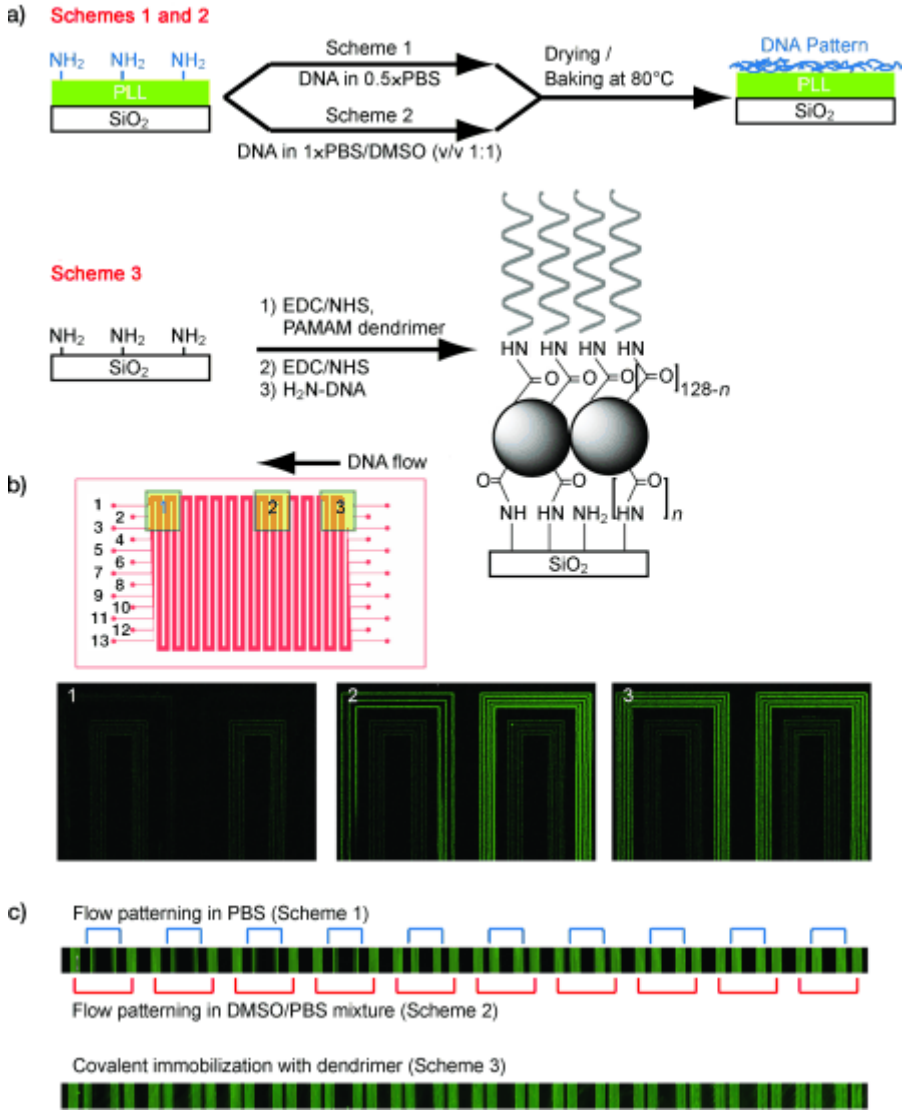


Figure 3.5.1 (a) Surface treatment schemes. **(b)** Design of the DNA patterning device (top) and fluorescence image of DNAs filled into the channel (still in solution). Outer five channels are filled with DNAs in 1:1 mixture of PBS and water (Scheme 1). Inner five channels are filled with DNAs in 1:1 mixture of PBS and DMSO (Scheme 2). Three channels in between are left empty for visualization. **(c)** Fluorescence images of patterned DNAs by three Schemes.

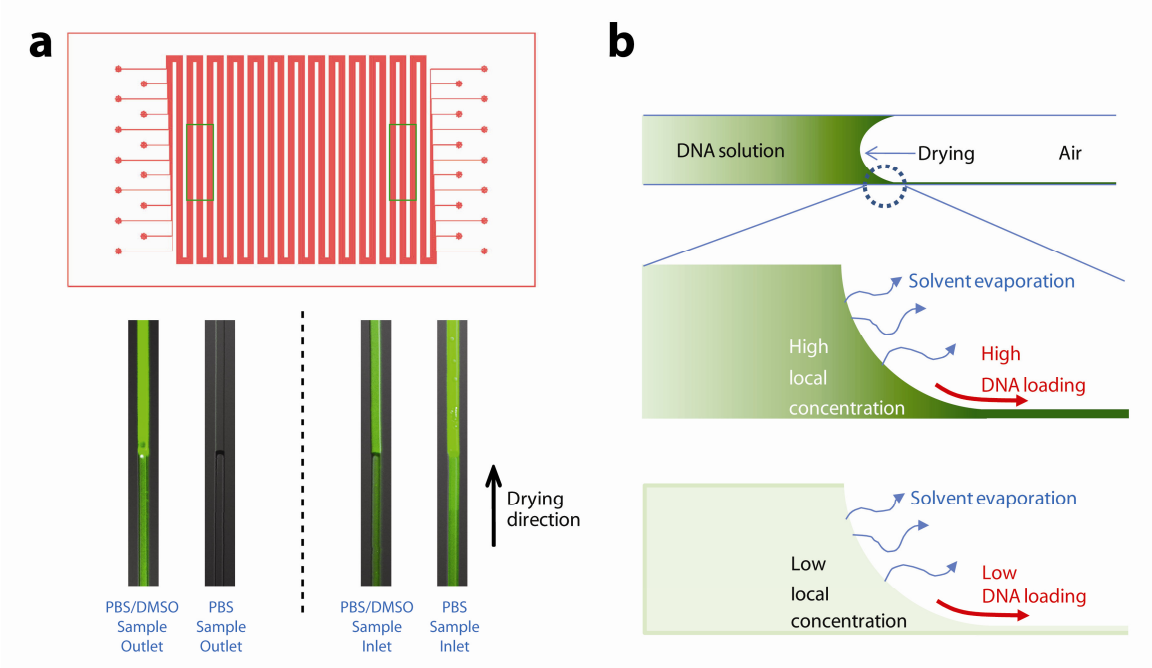


Figure 3.5.2 The microfluidic flow-patterning process to form the DNA barcodes. As solvent evaporates through the PDMS elastomer, the concentration of the DNA oligomer solution increases. The oligomers are eventually deposited on the microchannel surfaces. **(a)** Fluorescence images of DNA solutions during the drying process for Schemes 1 and 2, in the region of the receding meniscus, and near the outlet (left) and inlet (right) sides of the microchannel. Note that, at the inlet side, the fluorescence intensity near the receding meniscus is very high – evidence of the high local concentration of DNA due to solvent evaporation. The channel filled according to Scheme 1 exhibits no significant DNA near the channel outlet due to excessive electrostatically-driven depletion near the inlet side. The red arrow indicates the location of the meniscus. **(b)** Schematics for the drying process with different local concentrations. A high local concentration is required to achieve suitable DNA loading on the substrate.

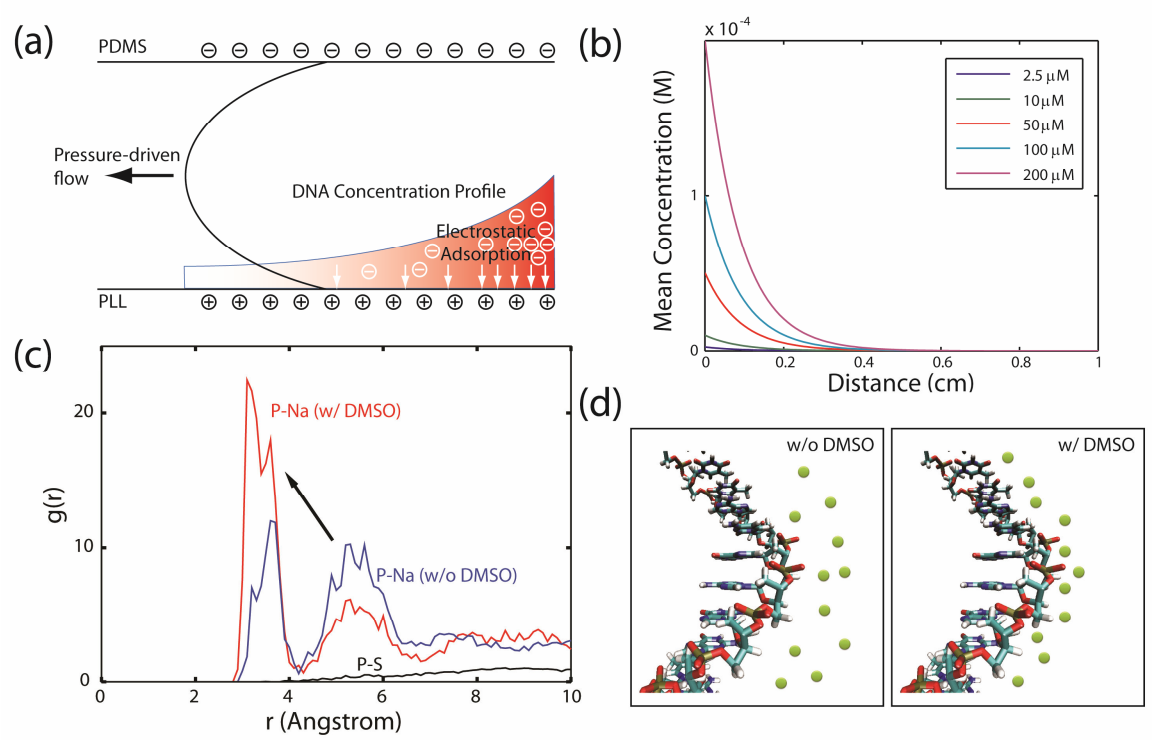


Figure 3.5.3 Electrostatic adsorption of DNAs on PLL surface and DMSO effect. **(a)** Schematic figure of the filling step. **(b)** Simulation result of electrostatic adsorption of DNAs to PLL surface. **(c)** Molecular simulation of DMSO effect: the radial distribution function of P atom of the phosphate group and the sodium ions. The presence of DMSO pumps sodium ions from the 2nd shell to the 1st shell (arrow). **(d)** Schematics for DMSO effect. Green circles represent sodium ions.

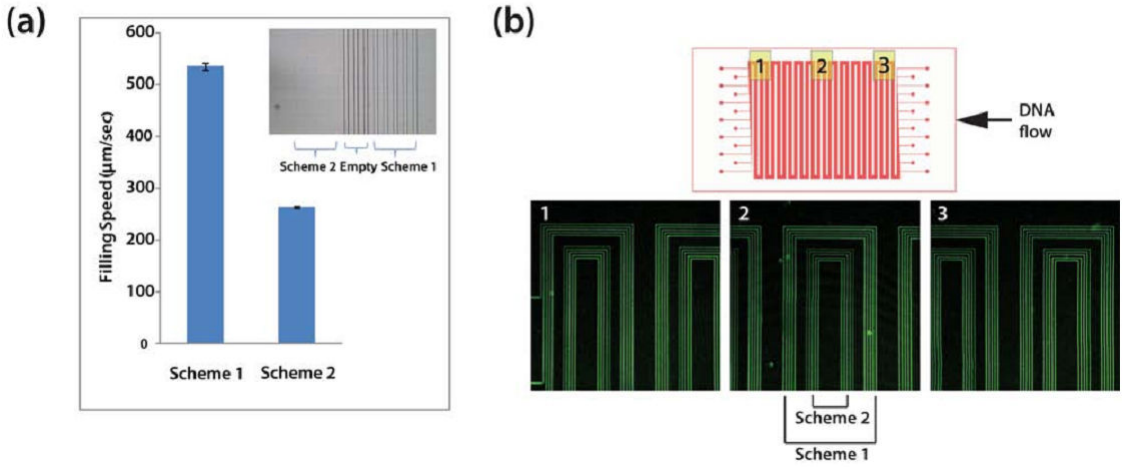


Figure 3.5.4 Results from experiments designed to more fully understand the effect of electrostatic adsorption of DNA within the microchannels during flow patterning. **(a)** Measurements of the flow speed of PBS solution of DNA oligomers (Scheme 1) and PBS/DMSO solution (Scheme 2) in the microfluidic channels. The filling process was optically monitored and recorded as a movie. The speed was calculated when the flow makes the fifth turn in the channel. The filling speed for Scheme 2 was less than that of Scheme 1, an observation that is attributable to the differential channel wetting between the two schemes (inset). The wetting of the PBS/DMSO Scheme 2 fluid was significantly better, a factor that minimizes bubble formation in the channel during the drying step. **(b)** Fluorescence images of DNA patterned within the microchannels of an O_2 plasma treated bare glass/PDMS device. The highly negative surface induced by plasma treatment minimizes electrostatic adsorption of DNA, resulting in uniform DNA distribution for both Scheme 1 and Scheme 2. The PDMS was solvent extracted just prior to bonding in order to prolong its hydrophilicity following plasma treatment. Panels 1, 2, and 3 represent different locations in the flow patterning device.

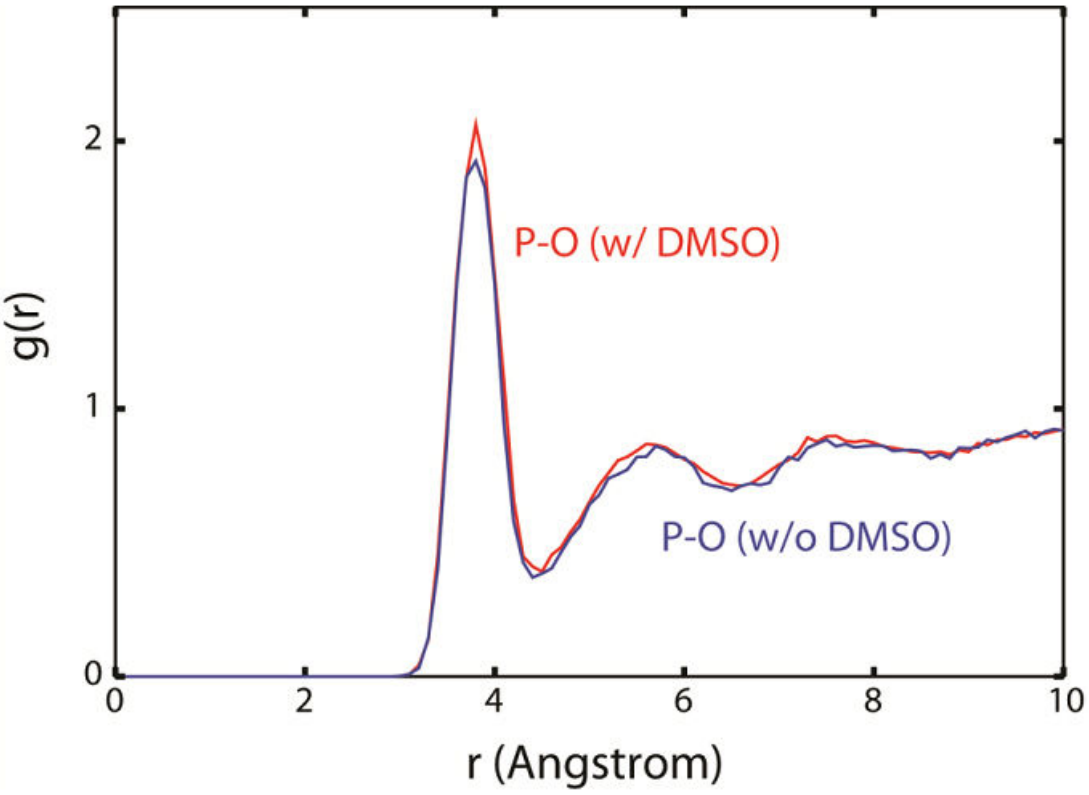


Figure 3.5.5 Molecular simulation result of the influence of DMSO in the Scheme 2 process. The radial distribution function of the P atom of the phosphate group of the DNA backbone and O atom of the water molecule is not influenced by the presence of DMSO.

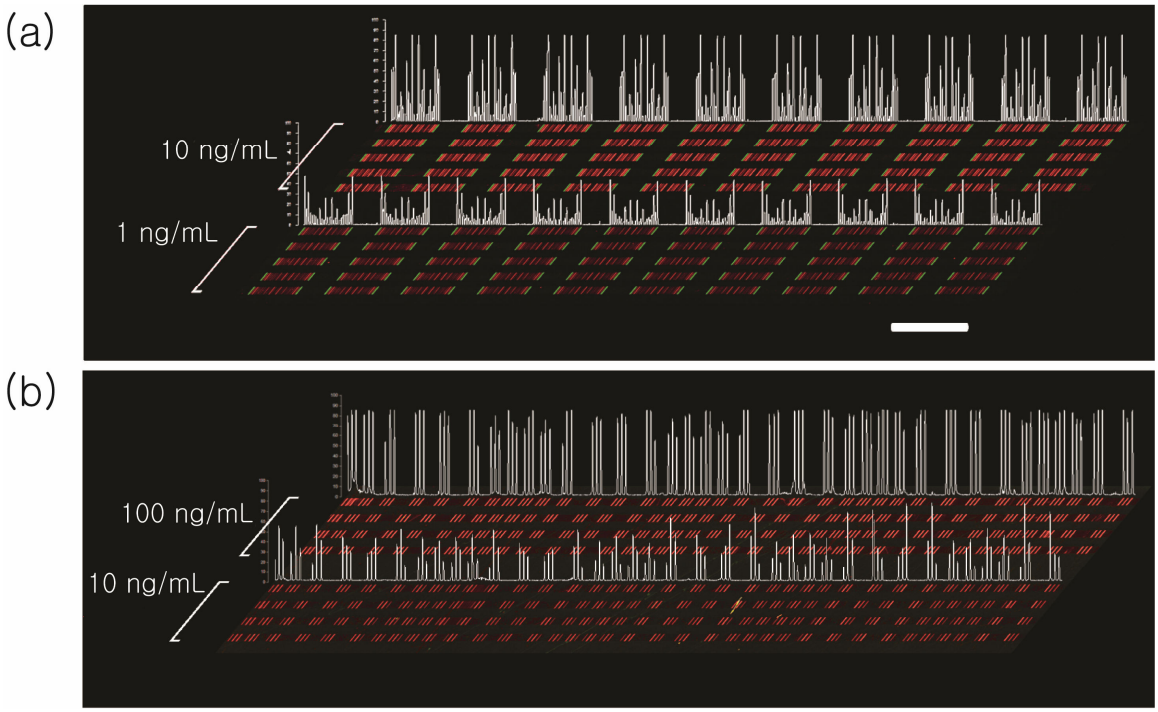


Figure 3.5.6 Raw data extracted from multi-protein calibration experiments performed on a substrate prepared by Scheme 2 **(a)** and Scheme 3 **(b)**. Signal intensity profiles sampled from one analysis channel per concentration are quantified in white. Scale bar: 2 mm.

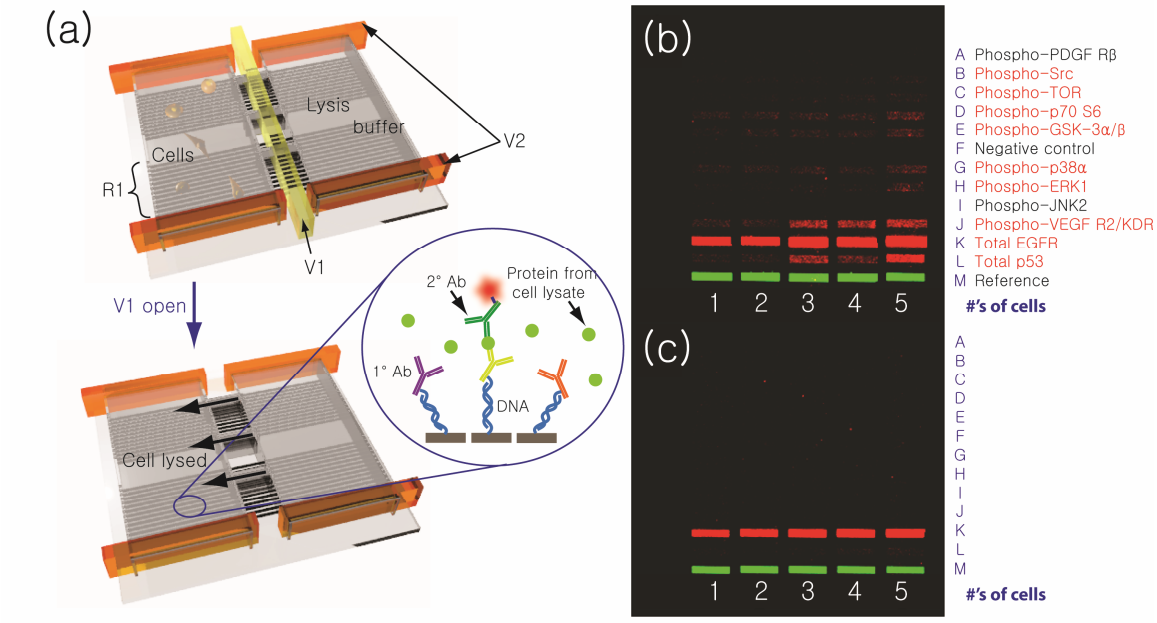


Figure 3.5.7 A schematic representation of the single-cell, intracellular protein analysis device. Single or few cells are incubated in an isolated chamber under varying stimuli. Intracellular proteins are assayed by introducing a pre-aliquoted lysis buffer, whereupon the released proteins bind to the DEAL (DNA-labeled antibody) barcode within the chamber. The benefits of utilizing a high-quality barcode are apparent when comparing data between substrates patterned using Scheme 2 **(b)** and substrates patterned using Scheme 1 **(c)**. Protein names listed in red font correspond to those which were detected using Scheme 2 barcodes.

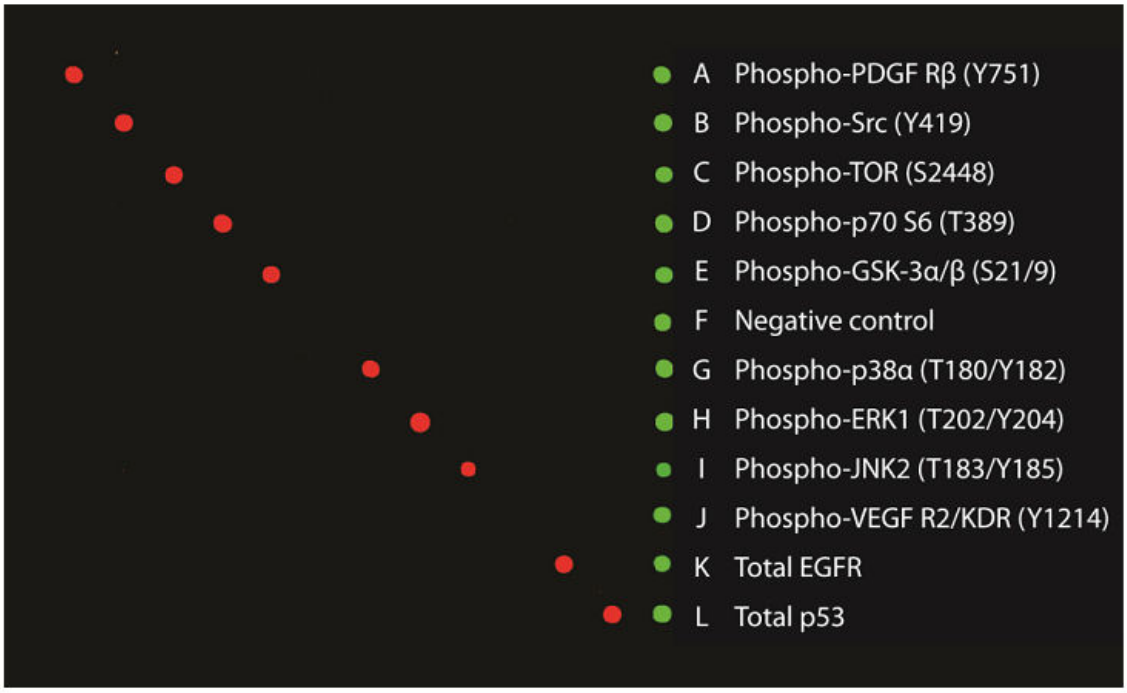


Figure 3.5.8 Antibody cross-reactivity tests. All antibodies were pre-selected based on such cross-reactivity tests. A pin-spotted DNA microarray was used for a DEAL-based protein detection approach - similar to the assays used within the SCBC microfluidic devices for single cell proteomics. Each row shows the results from different conditions. For all conditions, the same cocktail of DEAL conjugates was used, and included one conjugate for each of the 11 proteins assayed. For each row, only one target recombinant protein was tested. The target proteins were introduced at concentrations between 5-50 ng/mL, depending on the sensitivity of each antibody pair. Red spots are signals from the target proteins and the green spots are reference signals from Cy3-labeled DNA sequence M'. Phospho-VEGFR2 was not validated because the recombinant protein is not commercially available.

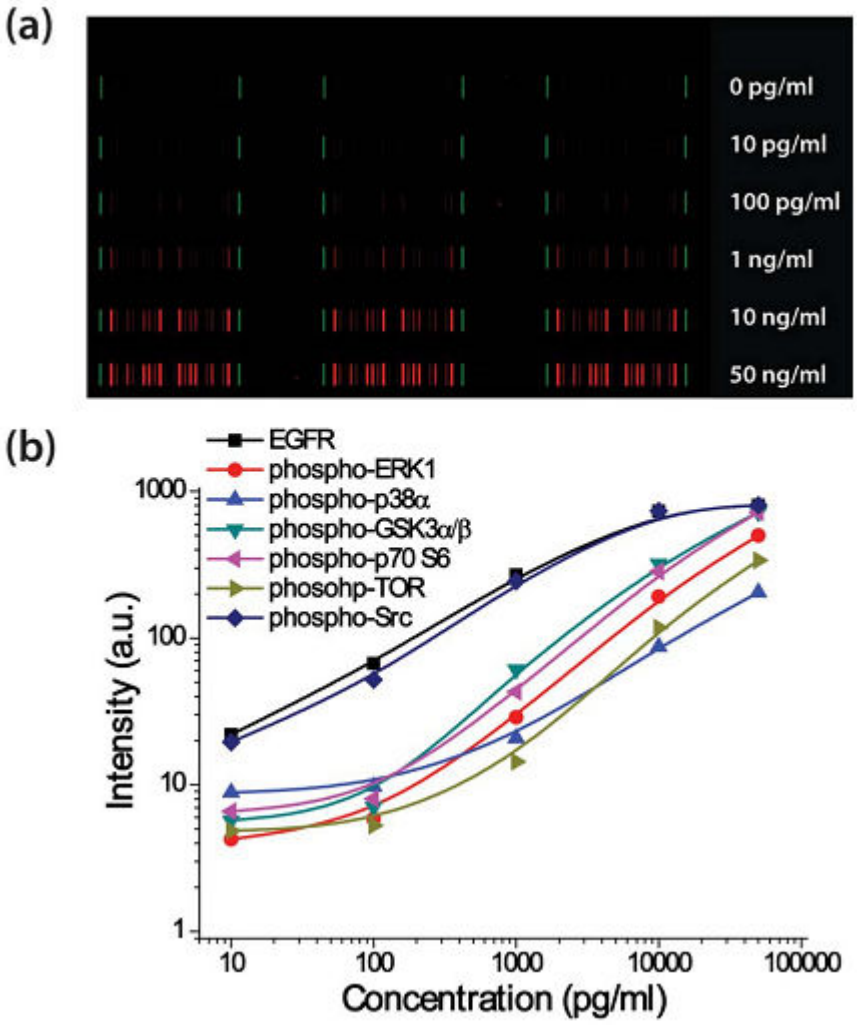


Figure 3.5.9 Calibration data for proteins in the panel. (a) Representative scanned images showing serial dilution measurements of selected proteins. Recombinant proteins were serially diluted (50 ng/mL, 10 ng/mL, 1 ng/mL, 100 pg/mL, 10 pg/mL and 0 pg/mL) in 1X PBS and flowed into the different microchannels of the microfluidic device for cell lysis analysis. Valves were immediately closed to compartmentalize standard proteins into microchambers followed by on-chip lysis buffer diffusion on ice for 2 hr. (b) Calibration curves of EGFR, p-ERK, p-p38α, p-GSK3α/β, p-p70S6K, p-mTOR and p-Src are plotted based on the results from a) to demonstrate the quantitative characteristics of the analysis. The sensitivities identified from the calibration curves are similar to standard ELISA sensitivities (e.g. EGFR: ~10 pg/mL, p-p70S6K: ~100 pg/mL, p-mTOR: ~200 pg/mL).

3.6 Tables

Name	Sequence	Tm °C (50mM NaCl)
A	5'- AAA AAA AAA AAA AAT CCT GGA GCT AAG TCC GTA-3'	57.9
A'	5' NH3- AAA AAA AAA ATA CGG ACT TAG CTC CAG GAT-3'	57.2
B	5'-AAA AAA AAA AAA AGC CTC ATT GAA TCA TGC CTA -3'	57.4
B'	5' NH3AAA AAA AAA ATA GGC ATG ATT CAA TGA GGC -3'	55.9
C	5'- AAA AAA AAA AAA AGC ACT CGT CTA CTA TCG CTA -3'	57.6
C'	5' NH3-AAA AAA AAA ATA GCG ATA GTA GAC GAG TGC -3'	56.2
D	5'-AAA AAA AAA AAA AAT GGT CGA GAT GTC AGA GTA -3'	56.5
D'	5' NH3-AAA AAA AAA ATA CTC TGA CAT CTC GAC CAT -3'	55.7
E	5'-AAA AAA AAA AAA AAT GTG AAG TGG CAG TAT CTA -3'	55.7
E'	5' NH3-AAA AAA AAA ATA GAT ACT GCC ACT TCA CAT -3'	54.7
F	5'-AAA AAA AAA AAA AAT CAG GTA AGG TTC ACG GTA -3'	56.9
F'	5' NH3-AAA AAA AAA ATA CCG TGA ACC TTA CCT GAT -3'	56.1
G	5'-AAA AAA AAA AGA GTA GCC TTC CCG AGC ATT-3'	59.3
G'	5' NH3-AAA AAA AAA AAA TGC TCG GGA AGG CTA CTC-3'	58.6
H	5'-AAA AAA AAA AAT TGA CCA AAC TGC GGT GCG-3'	59.9
H'	5' NH3-AAA AAA AAA ACG CAC CGC AGT TTG GTC AAT-3'	60.8
I	5'-AAA AAA AAA ATG CCC TAT TGT TGC GTC GGA-3'	60.1
I'	5' NH3-AAA AAA AAA ATC CGA CGC AAC AAT AGG GCA-3'	60.1
J	5'-AAA AAA AAA ATC TTC TAG TTG TCG AGC AGG-3'	56.5
J'	5' NH3-AAA AAA AAA ACC TGC TCG ACA ACT AGA AGA-3'	57.5
K	5'-AAA AAA AAA ATA ATC TAA TTC TGG TCG CGG-3'	55.4
K'	5' NH3-AAA AAA AAA ACC GCG ACC AGA ATT AGA TTA-3'	56.3
L'	5' NH3-AAA AAA AAA AGC CGA AGC AGA CTT AAT CAC-3'	57.2
M	5'-Cy3-AAA AAA AAA AGT CGA GGA TTC TGA ACC TGT-3'	57.6
M'	5' NH3-AAA AAA AAA AAC AGG TTC AGA ATC CTC GAC-3'	56.9

Table 3.6.1 Sequences and terminal functionalization of oligonucleotides: All oligonucleotides were synthesized by Integrated DNA Technology (IDT) and purified via high performance liquid chromatography (HPLC). The DNA coding oligomers were pre-tested for orthogonality to ensure that cross-hybridization between non-complementary oligomer strands was negligible (<1% in photon counts).

DNA Code	Antibody	Source
A'	Human p-PDGFRβ (Y751) kit	R&D DYC3096
B'	Human p-Src (Y419) kit	R&D DYC2685
C'	Human p-mTOR (S2448) kit	R&D DYC1665
D'	Human p-p70S6K (T389) kit	R&D DYC896
E'	Human p-GSK3α/β (S21/S9) kit	R&D DYC2630
G'	Human p-p38α (T180/Y182) kit	R&D DYC869
H'	Human p-ERK (T202/Y204) kit	R&D DYC1825
I'	Human p-JNK2 (T183/Y185) kit	R&D DYC2236
K'	Human total EGFR kit	R&D DYC1854
L'	Human total P53 kit	R&D DYC1043
J'	Capture: rabbit anti-human p-VEGFR2 (Y1214)	Abcam ab31480
	Detection: biotin-labeled mouse anti-human VEGFR2	Abcam ab10975

Table 3.6.2 Summary of antibodies used for cell lysis experiments: All antibody pairs except p-VEGFR2 were purchased as ELISA kits of R&D systems (DuoSet® Elisa Development Reagents) containing capture antibodies, biotinylated detection antibodies and standard proteins. Capture antibodies bind both phosphoryalted and unphosphorylated proteins. The biotinylated detection antibodies detect only the phosphorylated variants of the proteins. VEGFR2 capture antibody, p-VEGFR2 (Y1214) detection antibodies were purchased from Abcam.

3.7 References

1 Heath, J. R. & Davis, M. E. Nanotechnology and cancer. *Annu Rev Med* **59**, 251-265, (2008).

2 Heath, J. R., Phelps, M. E. & Hood, L. NanoSystems biology. *Mol Imaging Biol* **5**, 312-325, (2003).

3 Khalil, I. G. & Hill, C. Systems biology for cancer. *Curr Opin Oncol* **17**, 44-48, (2005).

4 Zheng, G., Patolsky, F., Cui, Y., Wang, W. U. & Lieber, C. M. Multiplexed electrical detection of cancer markers with nanowire sensor arrays. *Nat Biotech* **23**, 1294-1301, (2005).

5 Niemeyer, C. M. Functional devices from DNA and proteins. *Nano Today* **2**, 42-52, (2007).

6 Boozer, C., Ladd, J., Chen, S. & Jiang, S. DNA-Directed Protein Immobilization for Simultaneous Detection of Multiple Analytes by Surface Plasmon Resonance Biosensor. *Analytical Chemistry* **78**, 1515-1519, (2006).

7 Wacker, R., Schröder, H. & Niemeyer, C. M. Performance of antibody microarrays fabricated by either DNA-directed immobilization, direct spotting, or streptavidin-biotin attachment: a comparative study. *Analytical Biochemistry* **330**, 281-287, (2004).

8 Schroeder, H. *et al.* User Configurable Microfluidic Device for Multiplexed Immunoassays Based on DNA-Directed Assembly. *Analytical Chemistry* **81**, 1275-1279, (2009).

9 Douglas, E. S., Chandra, R. A., Bertozzi, C. R., Mathies, R. A. & Francis, M. B. Self-assembled cellular microarrays patterned using DNA barcodes. *Lab Chip* **7**, 1442-1448, (2007).

10 Bailey, R. C., Kwong, G. A., Radu, C. G., Witte, O. N. & Heath, J. R. DNA-encoded antibody libraries: a unified platform for multiplexed cell sorting and detection of genes and proteins. *J Am Chem Soc* **129**, 1959-1967, (2007).

11 Niemeyer, C. M., Sano, T., Smith, C. L. & Cantor, C. R. Oligonucleotide-directed self-assembly of proteins: semisynthetic DNA—streptavidin hybrid molecules as connectors for the generation of macroscopic arrays and the construction of supramolecular bioconjugates. *Nucleic Acids Research* **22**, 5530-5539, (1994).

12 Sano, T., Smith, C. L. & Cantor, C. R. Immuno-PCR: very sensitive antigen detection by means of specific antibody-DNA conjugates. *Science* **258**, 120-122, (1992).

13 Fan, R. *et al.* Integrated barcode chips for rapid, multiplexed analysis of proteins in microliter quantities of blood. *Nat Biotechnol* **26**, 1373-1378, (2008).

14 Dufva, M. Fabrication of high quality microarrays. *Biomolecular Engineering* **22**, 173-184, (2005).

15 Le Berre, V. *et al.* Dendrimeric coating of glass slides for sensitive DNA microarrays analysis. *Nucleic Acids Res* **31**, e88, (2003).

16 Bosman, A. W., Janssen, H. M. & Meijer, E. W. About Dendrimers: Structure, Physical Properties, and Applications. *Chemical Reviews* **99**, 1665-1688, (1999).

17 Benters, R., Niemeyer, C. M., Drutschmann, D., Blohm, D. & Wöhrle, D. DNA microarrays with PAMAM dendritic linker systems. *Nucleic Acids Research* **30**, e10-e10, (2002).

18 Angenendt, P., Glökler, J., Sobek, J., Lehrach, H. & Cahill, D. J. Next generation of protein microarray support materials:: Evaluation for protein and antibody microarray applications. *Journal of Chromatography A* **1009**, 97-104, (2003).

19 Ajikumar, P. K. *et al.* Carboxyl-Terminated Dendrimer-Coated Bioactive Interface for Protein Microarray: High-Sensitivity Detection of Antigen in Complex Biological Samples. *Langmuir* **23**, 5670-5677, (2007).

20 Kuo, A.-T., Chang, C.-H. & Wei, H.-H. Transient currents in electrolyte displacement by asymmetric electro-osmosis and determination of surface zeta potentials of composite microchannels. *Applied Physics Letters* **92**, 244102-244103, (2008).

21 Benn, J. A. *et al.* Comparative modeling and analysis of microfluidic and conventional DNA microarrays. *Anal Biochem* **348**, 284-293, (2006).

22 Comprehensive genomic characterization defines human glioblastoma genes and core pathways. *Nature* **455**, 1061-1068, (2008).

23 Kausaite, A. *et al.* Surface plasmon resonance label-free monitoring of antibody antigen interactions in real time. *Biochemistry and Molecular Biology Education* **35**, 57-63, (2007).

24 Krutzik, P. O., Irish, J. M., Nolan, G. P. & Perez, O. D. Analysis of protein phosphorylation and cellular signaling events by flow cytometry: techniques and clinical applications. *Clinical Immunology* **110**, 206-221, (2004).

25 Liang, Y. *et al.* Gene expression profiling reveals molecularly and clinically distinct subtypes of glioblastoma multiforme. *Proceedings of the National Academy of Sciences of the United States of America* **102**, 5814-5819, (2005).

26 Lee, J. C. *et al.* Epidermal Growth Factor Receptor Activation in Glioblastoma through Novel Missense Mutations in the Extracellular Domain. *PLoS Med* **3**, e485, (2006).

27 Thorsen, T., Maerkl, S. J. & Quake, S. R. Microfluidic large-scale integration. *Science* **298**, 580-584, (2002).

28 Duan, Y. *et al.* A point-charge force field for molecular mechanics simulations of proteins based on condensed-phase quantum mechanical calculations. *Journal of Computational Chemistry* **24**, 1999-2012, (2003).

29 Cornell, W. D. *et al.* A Second Generation Force Field for the Simulation of Proteins, Nucleic Acids, and Organic Molecules. *Journal of the American Chemical Society* **117**, 5179-5197, (1995).

30 Plimpton, S. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics* **117**, 1-19, (1995).

31 Tung, C. S. & Carter, E. S. Nucleic-Acid Modeling Tool (NAMOT) - An interactive graphic tool for modeling nucleic-acid structures. *Comput. Appl. Biosci.* **10**, 427-433, (1994).

32 Jorgensen, W. L., Chandrasekhar, J., Madura, J. D., Impey, R. W. & Klein, M. L. Comparison of simple potential functions for simulating liquid water. *The Journal of Chemical Physics* **79**, 926-935, (1983).

33 Hockney, R. W. & Eastwood, J. W. *Computer Simulation Using Particles*. (McGraw-Hill, 1981).

Chapter 4

A Robotics Platform for Automated Batch Fabrication of High Density, Microfluidics-Based DNA Microarrays, with applications to single cell, multiplex assays of secreted proteins.

4.1 Introduction

Miniaturized DNA and antibody arrays, when incorporated into microfluidics environments, provide an appealing technology for multiplexed detection of biological molecules from small biospecimen samples, such as tissue extracted from skinny needle biopsies, pinpricks of blood, or single cells. Traditional robotics spotted DNA arrays are typically characterized by 150 µm spot sizes, patterned at ~300 µm pitch, which limits their use in many microfluidics-based applications. However, those arrays can be produced in both high throughput and high quality, with typical spot variation ranging from 5-10% across individual substrates, and 10-30% between substrates[1-3]; this makes spotted arrays widely available for a host of standard biological assays. Approaches towards the production of miniaturized DNA arrays include photolithography[4, 5], dip-pen lithography[6, 7], microfluidics-based flow-patterning[8-13], and, for relatively simple arrays, microstamping[14-16]. We have utilized microfluidics-based flow patterning to prepare

DNA barcodes over entire glass substrates, with stripes of 20 µm width at a pitch of 40-50 µm. This represents an approximately 10-fold increase over what can be achieved using spotted arrays. When these stripes are entrained in designed microfluidics chips, they can be converted into high density antibody arrays[17-20] for multiplexed assays from few nanoliter volume biological samples[21], or even single cells[22]. However, miniaturized DNA barcode preparation is done manually, and so are limited in their use. We report here on a robotics platform which enables relatively high throughput production of miniaturized DNA barcode arrays. The platform achieves high barcode uniformity across the surface of a single chip, and high reproducibility from chip-to-chip. We demonstrate the value of the technology by applying it towards a platform designed to simultaneously assay a panel of eleven secreted proteins from single cells, with >1000 single and few-cell assays executed in parallel on a single microchip.

The challenges associated with automating the production of flow patterned barcodes are two-fold. First, there is the problem of scale. The barcodes are initially patterned as ~0.8m long, 20 µm wide stripes of ssDNA that meander over the surface of an aminated or poly-L-lysine-coated glass slide. Most applications yield improved performance as the DNA loading within a given stripe is increased[19], and so it is important to maximize both high and uniform loading across the entire length of these channels. The aspect ratio of each stripe (10^5-10^6), coupled with the loading requirements, places severe demands on the flow patterning chemistry, which has been recently optimized[22]. In addition, a full barcode pattern is comprised of between 10 and 50 stripes, each of which represents a distinct ssDNA sequence. Thus, a robotics system

must self-align a large number of injectors with a given elastomeric mold at an alignment precision of order 100 μm, and it must do so multiple times across a ~900 cm^2 area, in order to sequentially and automatically address many chips.

The second challenge relates to the mechanical characteristics of the elastomeric flow patterning mold. This mold is only weakly bonded to the glass surface; it must be removed once the barcoding process is complete. In addition, the individual stripes within a barcode are separated from one another by as little as 20 μm, which is the wall thickness of microchannels in the flow patterning template. Thus, the machine's injector head must mate and then disengage each flow patterning elastomeric chip very gently, and the intermediate DNA injection process must be executed at low pressures to prevent both wholesale elastomer delamination and localized channel-to-channel delamination, both of which lead to chip failure.

We first give a brief overview of the robotics-driven sequential production of up to 18 barcoded glass slides, followed by a statistical evaluation of the quality of the barcoded slides - both in terms of barcode variability on a given slide, and across different slides produced in the same run. We then discuss the application of these barcoded substrates towards multiplex assays of secreted proteins from single cells.

4.2 Experimental Section

4.2.1 Robotics design.

The robotics-driven flow patterning apparatus is shown in Figure 4.1A. Major components of the robotics are numerically labeled in the figure, including the chip support tray (#1), the injector module (#2), the DNA solution reservoirs (#4), and the translation motors (#5). A detailed scheme of the injector module is presented in Figure 4.1B; the injector employs a standard microfluidics interfacing scheme wherein stainless steel pins are inserted into punched access holes that bridge the top surface of the PDMS flow patterning molds with the microchannel/glass surface interfaces below[23, 24]. The stainless steel pins are embedded within a laser-drilled acrylic "injector plate", and are arranged according to a pre-determined pattern that matches the substrate access holes (Figure 4.1C). This scheme allows for a high density of fluidic inputs, and it reduces substrate filling to a parallel process. However, the scheme also introduces a challenge related to the alignment of the pins to the access holes: the pins are 650μm in diameter while the access holes are only 500μm. The dimensional mismatch forces the elastomer to expand upon interfacing and thereby form a leak-proof seal around the pins. However, the soft nature of the elastomer also means that misalignment of the two components can lead to unwanted deformation or unintended puncturing of the PDMS instead of smooth mating of pin and hole. The problem is compounded by the

fact that all the pins must be aligned across simultaneously, leaving very small tolerances for angular misalignment.

As such, substrate-injector alignment is done in two phases. A pre-alignment is achieved by virtue of the plastic cutouts on the substrate tray, which loosely define the locations of the (up to) 18 PDMS flow patterning chips. Finer alignement is provided by a Cognex IS5400 camera system mounted to the side of the injector head. Prior to engaging each substrate, the camera is positioned directly over the chip and images the access hole pattern, comparing it to a pre-trained image using built-in pattern recognition algorithms (Figure 4.1C); *x*, *y*, and θ deviations are reported to the control software which adjusts the appropriate translation stages and re-images the substrate iteratively until a null deviation reading is achieved. The injector head is then shifted a pre-calibrated distance to align with the substrate and is slowly lowered into place until the pins sink 1 mm into their corresponding access holes.

Once engaged, DNA solutions are supplied to the injector head from a set of adjacent, disposable microvials via short lengths of Tygon tubing. The delivery of precisely metered, microliter scale aliquots, is typically accomplished by external syringe pumps, but we offload metering responsibility to the PDMS chips themselves. Specifically, the microfluidic channels were fabricated with a set of input access holes, but no output holes, thereby creating a closed system upon substrate engagement. Because PDMS is air permeable, a pressurized solution injected into the input ports can displace air within the microchannel until it reaches the end [25]. In this way, a very precise volume, defined by the input access hole and microchannel dimensions, is metered into each channel.

The on-chip metering allows for a relatively simple implementation of the pressure system used to drive solutions, as depicted in Scheme 1. Briefly, compressed air is first regulated to the pressure required to drive solutions through their microfluidic channels. Typical pressures range from 2-5psi, and are set with inverse proportion to the pattern density of the chip in order to prevent cross-contamination of solutions from adjacent channels. A proportional valve conveys the pressure via a gentle ramp to avoid splashing the solutions in their microvials downstream, and a pair of three-way solenoid valves integrates this pressure line to establish one of three pressure states in the microvials: positive pressure, vented, and closed. After engaging a substrate, the microvials are gently pressurized to drive their solutions into the microchannels. Once filling is complete, the solenoids reconfigure to vent the microvials, and then reconfigure again to create a closed system prior to disengaging. This final state helps to balance hydrostatic pressures and prevent leaking from the injector pins in the disengaged state.

Disengaging the injector head from a substrate requires additional engineered components: because the injector pins form a tight seal with their corresponding substrate access holes, care must be exercised to prevent the PDMS mold and glass substrate from delaminating while extracting the pins. The injector plate is therefore fitted with four pneumatic pistons whose rods secure a second “pressure plate” to its underside. Matching through holes in the pressure plate enable the injector plate pins to protrude beneath it during substrate engagement and manipulation. When disengaging a substrate, this pressure plate is extended to brace the PDMS firmly

against the slide tray while the pins are extracted. The entire injector/pressure plate assembly slides into a slot on the machined injector carriage and is reproducibly located via two shoulder screws. This modular implementation makes it easy to swap injector heads with different pin configurations from run to run, thereby allowing significant flexibility in substrate design.

4.2.2 Substrate fabrication.

We standardized virtually all aspects of the microfluidic flow channel chip dimensions, and streamlined their production[26]. To generate PDMS substrates, a deep reactive ion etched (DRIE) Si master is clamped between two machined aluminum plates; the upper plate contains cutouts that will define the substrate dimensions, and the master is positioned such that its features are aligned within these cutouts (Figure 4.2A/B). The resulting dimensional uniformity, particularly in thickness, obviates the need for sophisticated depth control when interfacing the injector head with substrates; a pre-calibrated constant is sufficient. The most critical substrate features, however, are the access holes which bridge the microfluidic channels with the top side of the substrate. We developed a template to mold the access holes as the substrate cures. Specifically, stainless steel wires are embedded into a laser-drilled acrylic plate in the required pattern. After pouring PDMS into the aluminum/silicon mold assembly (Figure 4.2A), this plate is secured to the top side such that the wires extend into the PDMS below. Upon curing, the plate is removed, leaving behind the templated inlet and outlet ports. The wires do not extend completely to the underlying Si mold, which prevents damage

to the Si mold. Thus, a very thin membrane of PDMS at the bottom of each access hole is retained. For the inlet ports, these membranes are easily punctured in a single step by pressing the substrate onto the top side of the same acrylic plate used to originally mold the holes. The thin membranes are retained in the outlet ports. This means that we are able to generate a dead-end fill substrate that yields extremely consistent metering volumes and also fills relatively quickly due to the high air permeability of the thin membranes at the output half.

4.2.3 Software and Operation.

The instrument's mechanical components are all controlled by custom software written in National Instruments Labwindows/CVI. Stage motion is powered by a standard 6K 4-Axis Motion Controller, while a NI DAQ card (PCI-6052E) provides digital and analog outputs to regulate an array of relays, solenoids, and proportional valves. The software presents an interface that allows users to click which of the 18 microchip positions on the substrate tray are to be processed. Once a run is initiated, the software assumes active control of all components, and processes the marked substrates sequentially without further intervention. Scheme 2 illustrates the instrument's process flowchart for a typical barcoding run; from the user perspective, it simply consists of laying out the substrates in their tray, filling the microvials with DNA solutions, and loading the appropriate configuration files before pressing a button to start the run. As such, the user can pattern up to 18 barcode substrates with < 1 hr setup time, which is ~20-fold faster than the manual process, and at least competitive with DNA spotter tools. When

automated filling is complete, the barcode microarrays are finalized using the same standard protocols employed for spotted slides: after a 24 hour incubation period, the slides are given a short UV exposure of ~1800 mJ/cm^2 which crosslinks the DNA in place. Following an additional 12 hour incubation, the PDMS is removed, the substrates are rinsed, and are then ready for use[27].

4.3 Results and Discussion

4.3.1 Pattern fidelity and chip-to-chip consistency

We prepared a set of six 20-channel PDMS barcode substrates featuring 20μm channels at 120μm pitch. For each chip, 4 adjacent microchannels were utilized to pattern 4 unique ssDNA strands, denoted A-D (SI, Table 1). To analyze the fidelity and amount of channel-to-channel leakage that occurred during patterning, the barcoded chip was effectively split in half, and 2 non-adjacent stripes were assayed on one half of the chip, while the remaining two were assayed on the other half. In this way, if DNA from channels A or C leaked into either or both of channels B or D, for example, such leakage would be detected. The DNA stripes on the first five of the substrates were investigated by first blocking with 1% BSA and then incubating with Cy3-conjugated complementary DNA. Figure 4.3A depicts the raw signal from one of these substrates. Only the intended four channels exhibit signal in a repeating fashion across the chip, and the automated process did not lead to delamination of the PDMS from its glass substrate.

We quantified the fluorescence signal from each of the five barcode-patterned substrates to assay fore quality and consistency. Raw fluorescence intensities were collected for each DNA sequence at eleven locations per chip, spanning a 28 cm length of the flow channel. Figure 4.3b compares the averaged intensities for each stripe on each chip. The error bars reveal high signal uniformity across the eleven imaged regions, and the absolute signal intensities for each DNA strand are in good agreement with one another across the five chips. DNA stripes on individual chips consistently demonstrate <10% coefficient of variation (CV), while the averaged values for each DNA across multiple chips exhibit < 12% CV (Figure 4.3C). These data confirm that the automated instrument is capable of generating a high quality and consistent batch of substrates, with a quality that is comparable to previously established standards for hand-made substrates[22].

4.3.2 Single cell secretion studies

We now turn towards demonstrating the applicability of our barcoded substrates to miniaturized bioassays via the mutiplex detection of proteins secreted from single macrophage cells. The barcoded glass slides are first incorporated into a microfluidics chip, called a Single Cell Barcode Chip (SCBC) designed for the capture of single cells and small cell colonies (Figure 4.4). Microfluidic chip designs for cell isolation and interrogation have previously been reported albeit with different detection schemes[28-30]. Here, the DNA barcodes are then converted into antibody barcodes using a cocktail of DNA-labeled capture antibodies[19]. Cells are then introduced and isolated into any of

approximately 1000 separate 3 nanoliter volume microchambers on the SCBC. The numbers of cells in a given chamber are recorded; of the 1000 such experiments on a single chip, typically 100-200 are single-cell experiments, while the remaining are 0-cell, 2-cell, 3-cell, etc., experiments (Figure 4.4B). The chip is then incubated for a period of time during which the captured cells secrete proteins that are selectively captured by the antibody barcodes. The cells are then washed from the chip and a cocktail of detection antibodies and fluorescent dye labels are added to develop the protein assays (Figure 4.4B,C). The measured fluorescence levels from the individual barcode stripes are digitized and then compared against calibration curves (Figure S1) to provide an estimate of the numbers of protein molecules detected, which, in turn, yields information on the sensitivity of the robotics-patterned antibody barcodes. By comparing the statistics of protein secretion levels from single cells assayed on one chip with identical assays from a second chip, the chip-to-chip variability can be assessed. A related SCBC, but designed for assaying phosphorylated membrane and cytoplasm proteins from single, lysed cancer cells, has been recently reported by us[22]. That chip utilized hand-made barcode patterns, and only permitted ~120 single- and few-cell experiments per chip, but was otherwise similar in concept to the chip described here.

The DNA barcodes utilized for this demonstration were 20-element arrays. Twelve of the elements were used for the bio-assay, and each contained one of 12 unique DNA sequences, A-M (SI, Table S1), flow-patterned with a stripe width of 20µm and at a 50µm pitch. Two substrates from a batch of four were carried forward for the cell assays; the barcoding PDMS was removed and the new microfluidic device (Figure

4.4A) was bonded in its place. Eleven of the DNA stripes were converted to form an antibody array, with antibodies chosen to correspond to secreted proteins. They included: Monocyte Chemoattractant Protein (MCP)-1, Interleukin (IL)-6, Granulocyte Macrophage Colony Stimulating Factor (GMCSF), Macrophage migration Inhibitory Factor (MIF), Interferon (IFN)-γ, Vascular Endothelial Growth Factor (VEGF), IL-10, IL-8, Matrix MetalloPeptidase (MMP)-9, and Tumor Necrosis Factor (TNF)-α, and IL-2. IL-2 is not expected to be secreted from macrophage cells, and so serves as a negative control. The remaining DNA stripe provides an alignment reference for the final read-out. Once the antibody array is assembled, the cells were prepared for loading. We investigated the human monocyte cell line, THP-1. These cells were first differentiated into the macrophage lineage using phorbol 12-myristate 13-acetate (PMA) and stimulated with lipopolysaccharide (LPS) [31] and then loaded, as a dilute suspension, into the 80 microchannels that span the length of the barcoded glass slide. The PMA elicits a morphological change in the THP-1 cells (Figure S2), and LPS activates the Toll-like Receptor-4 on the cell surface [32, 33], emulating the response of macrophages to gram negative bacteria. A set of 14 integrated valves[34] are activated to divide the microchannels into 1040 discrete microchambers, each containing single or small numbers of cells (Figure 4.4A). Each chamber is examined to record the number of cells it contains, and the entire platform is then placed in a CO_2 incubator at 37°C for 24 hours while secreted proteins are recorded onto the antibody microarray. Afterwards, the cells are flushed from the microchannels, and the protein assays are developed with a cocktail of biotinylated secondary antibodies followed by the addition of Streptavidin-

Cy5 fluorophores. The fluorescence intensities are digitized through the use of an Axon GenePix 4400A array scanner, coupled with custom written image processing routines. The resulting data is a table that lists, for each microchamber, the numbers of cells in that microchamber, and the digitized fluorescence for each of the assayed proteins and the DNA alignment reference stripe.

We quantified our raw data by first establishing a signal baseline. For each protein assayed, we averaged the raw signal values across all the individual chambers which contained single cells. Using the averaged signal recorded from the IL-2 assay stripes as the noise level, we calculated the signal-to-noise (S/N) for each protein, and set a threshold of S/N ≥ 4 to signify positive detection of a protein. By that standard, nine proteins were identified (S/N levels are in parentheses after the protein names): IL-6 (4), INF-γ (14), GMCSF (27), VEGF (89), IL-10 (190), MMP9 (498), IL-8 (560), TNF-α (566), and MIF (1504). Comparisons against separately generated calibration curves (Supplementary Fig S1) revealed limits of detection that were similar to or slightly worse than commercial ELISAs. For example, VEGF yielded 3 pg/mL vs 2.5 pg/ml and IL-8 yielded 75pg/mL versus 25 pg/mL.

Full analysis of the single cell secretome data is beyond the scope of this paper, and so we simply provide some preliminary analysis of the data in order to validate the barcode and chip technology. Figure 4.4B depicts a set of four adjacent chambers, two of which contained single cells and two of which contained four cells each. We grouped the (background-subtracted) data according to numbers of cells per microchamber. The data within each group was then sorted according to the level of MIF secretion. Error

bars are plotted for many of the data points; these are derived from chambers that contained two copies of the 20-element barcode and thereby yielded replicate measurements from which measurement error could be estimated. The plot of Figure 4.5A clearly demonstrates a cumulative effect in the observed signal, as increasing numbers of cells yields a greater proportion of chambers with high signal levels. The maximum signal level for each of the shown set of experiments is near saturation and so does not increase with increasing numbers of cells. Note that a percentage of the n=1, 2, and 3 cell chambers yield MIF signal levels that are similar to those observed for the 0-cell chambers: 52% (1 cell), 37% (2 cell), 22% (3 cell). These values consistently indicate that between 50 and 60% of the individual macrophage cells secrete low levels of MIF, but they also indicate that the 1, 2, and 3 cell data sets represent sets of measurements that are distinct from each other, and distinct from the 0-cell data. There was no correlation between the level of secreted protein and the location of the associated microchamber on the chip surface.

A heat map of protein secretion levels for the single cell experiments is provided in Figure 4.5B. This data demonstrates the stochastic (and not unexpected[35-37]) nature of protein expression at the single cell level. These single cell fluctuations, for a given protein level, can be compared between two chips as a means of comparing the chip-to-chip variability. While any given microchamber may yield a very different result from another microchamber, a statistically significant measurement of the single cell fluctuations, as recorded on one chip, should be indistinguishable from those recorded on a second chip. Such a comparison is provided in Figure 4.5C between data sets

collected from two different chips, and both the average protein level and the detailed distributions prove to be chip-independent. Similar analyses were also done for the proteins MMP9 ($p=0.640$) and TNF- α ($p= 0.435$) (Figure S3); among the remaining proteins, only IL-8 & IL-10 fail to yield p-values above 0.1. Of these two proteins, time-studies (not presented here) indicate that the secretion of IL-10 is delayed relative to the other proteins, and so longer time studies would likely increase the chip-to-chip p-values for this protein. IL-8, while secreted early, is also characterized by ca. 10x higher background signal, and so is intrinsically a less reliable measurement than the other detected proteins. The results indicate a high level of consistency across both a single microchip, and across multiple chips. This means that data taken from different chips could be seamlessly integrated to increase sampling statistics.

4.4 Conclusions

Traditional DNA microarray technology has proved an exceedingly useful tool, thanks in part to the development of significant infrastructure dedicated to microarray production and processing. As the applications of microarrays continue to evolve, there is a strong march towards further miniaturization. Alternative technologies such as dip-pen lithography[6, 7], electrohydrodynamic jet printing[38], and spotting with custom-built, nanostructured spotting pins[39], are pushing the low-micron to sub-micrometer patterning regime. Although microfluidic flow patterning does not yet extend to sub 10-micrometer features, it provides an attractive combination of multiplexing,

miniaturization, and throughput. The approach also permits flexible spot morphology and requires readily-accessible materials that allow cost-effective, in-house chip fabrication. However, in order to match the convenience and availability of spotted microarrays, an automated solution is needed to generate flow patterned substrates. The approach described here implies that microfluidics-flow patterned substrates can be reliably and reproducibly prepared using robotics-based automation, and that the resultant antibody barcode arrays exhibit assay characteristics that are at least as good as those prepared using standard, spotted arrays.

4.5 Figures

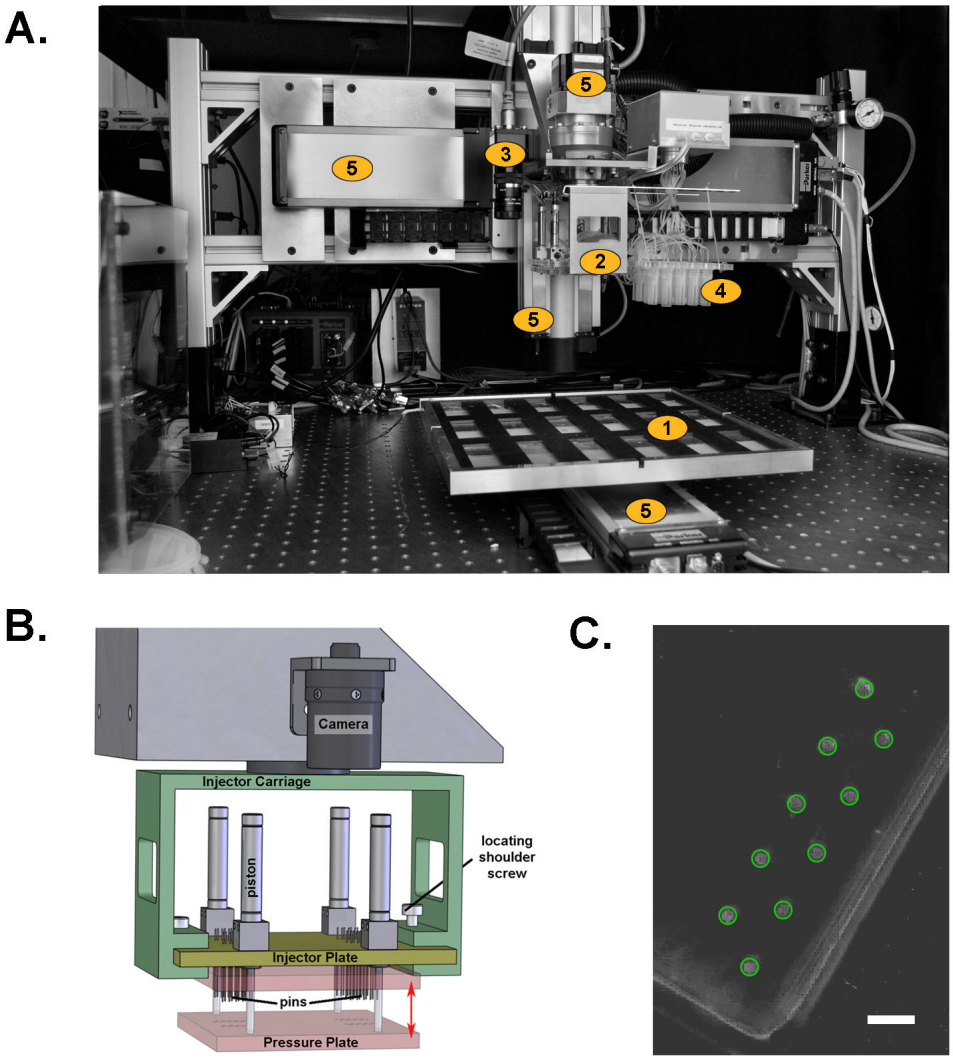


Figure 4.5.1 (A) An overview of the instrument as implemented. Substrates are arrayed on the slide stage (1) and thereafter are addressed sequentially by a mobile injector head (2). A camera system (3) images the substrates' access holes to guide alignment as the injector interfaces each substrate, and reagents are supplied from a set of adjacent microvials (4). Mechanical motion in the x, y, z, and θ axes is effected by a combination of linear stages and stepper drives (5). **(B)** Schematic detail of the injector assembly illustrates the pin interface used to engage each substrate and the pneumatic pressure plate which prevents delamination when disengaging. **(C)** Sample image from the camera system during substrate alignment. The field of view encompasses just one corner of the substrate; green circles (enhanced for clarity) indicate access holes in the PDMS that have been recognized by the software's pattern recognition algorithm and are used to finely adjust the injector head prior to interfacing. Scale bar: 2mm

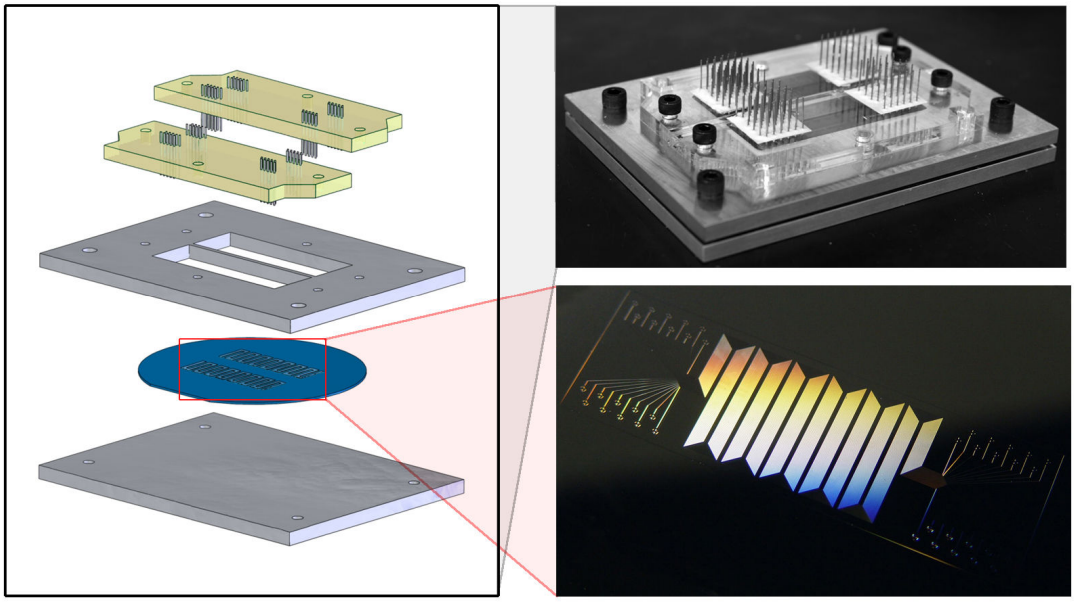
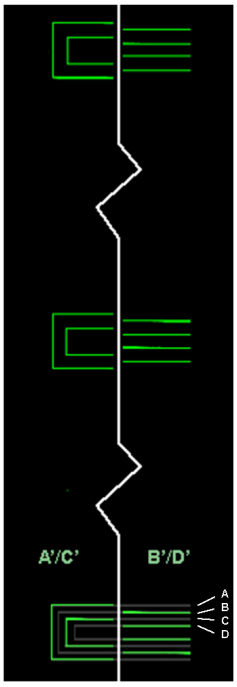
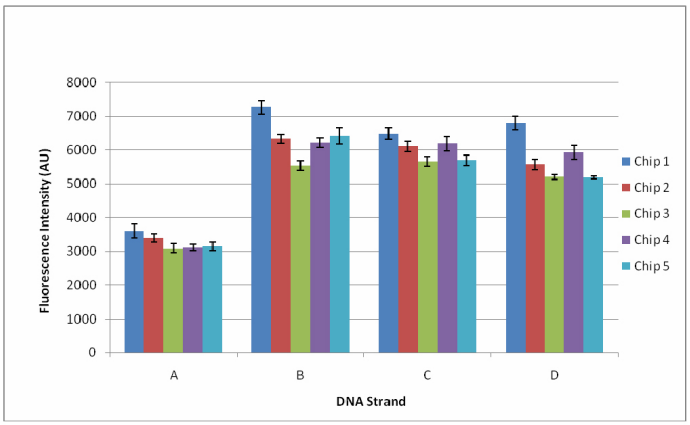


Figure 4.5.2 The aluminum stencil used to fabricate each PDMS substrate standardizes overall dimensions and access hole placement and size. A silicon wafer bearing barcode microfeatures is first sandwiched between two aluminum plates; PDMS precursor is poured into the cutouts and acrylic plates for molding access holes are affixed on top.

A



B



C

	A	B	C	D
Chip 1	2.81%	4.01%	1.32%	4.77%
Chip 2	7.76%	4.37%	5.02%	5.40%
Chip 3	9.24%	5.35%	5.33%	2.74%
Chip 4	5.49%	4.79%	6.68%	6.80%
Chip 5	8.36%	7.19%	5.25%	1.49%
All Chips	6.69%	9.63%	5.76%	11.65%

Figure 4.5.3 (A) Fluorescence image of a barcode microarray that is validated with alternating DNA sequences to check for unintended crosstalk or contamination. The channel morphology is overlaid on the bottom repeat; each microchannel meanders across the chip to create multiple repeats of the same pattern. **(B)** Average fluorescence intensities for DNA sequences A-D are quantified for five separate barcode substrates patterned by the instrument. Error bars represent the standard deviation calculated from eleven measurements per sequence. **(C)** coefficients of variance for each DNA sequence calculated from eleven regions of each chip. The averaged intensity for each sequence is then compared amongst chips and a CV is calculated to quantify chip-to-chip consistency.

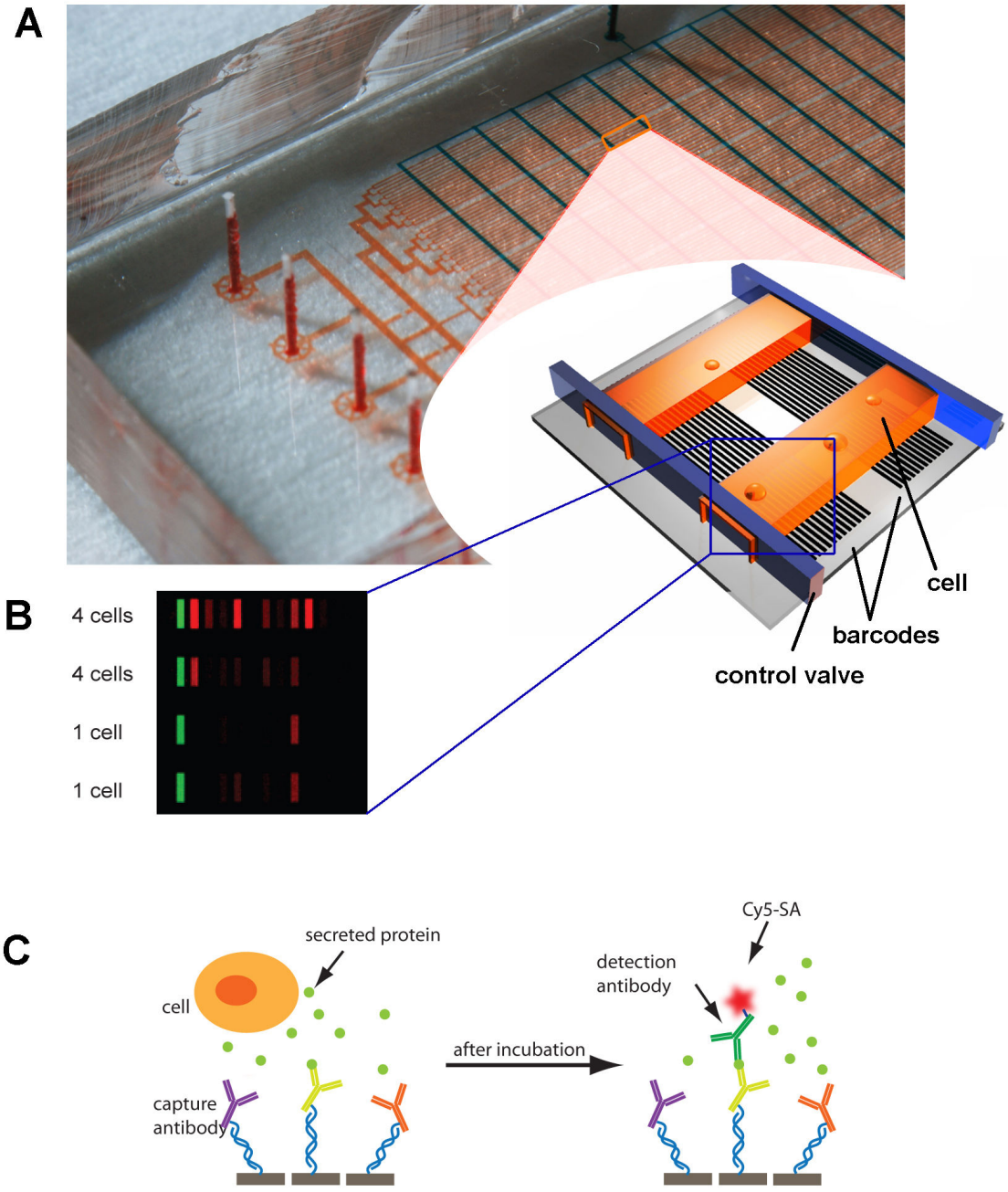


Figure 4.5.4 (A) Optical micrograph of the single cell secretion microfluidics, with schematic inlay of two discrete chambers. Raw data cropped from four adjacent chambers is depicted in **(B)**; the green bar is used for registration while red bars represent protein data. **(C)** Schematic representation of single cell secretion experiments. Capture antibodies are arrayed onto a barcode microarray via DEAL chemistry and sequester proteins secreted from an adjacent cell. The assay is developed by flushing with detection antibodies and a fluorescent reporter that form an ELISA-like sandwich.

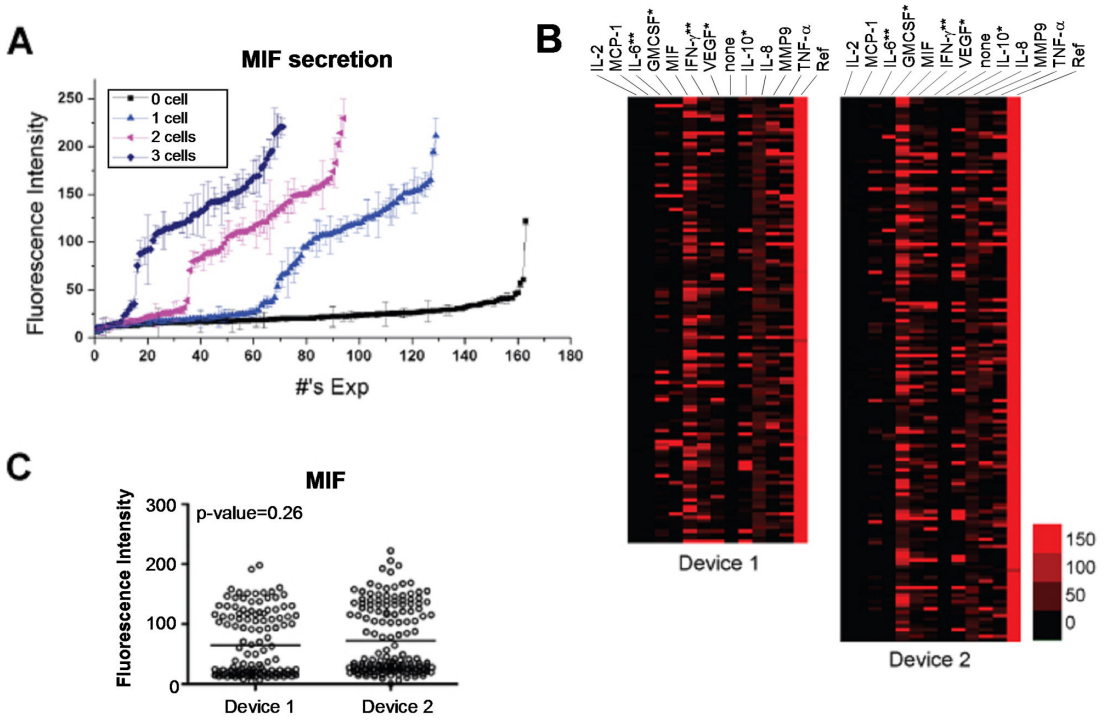
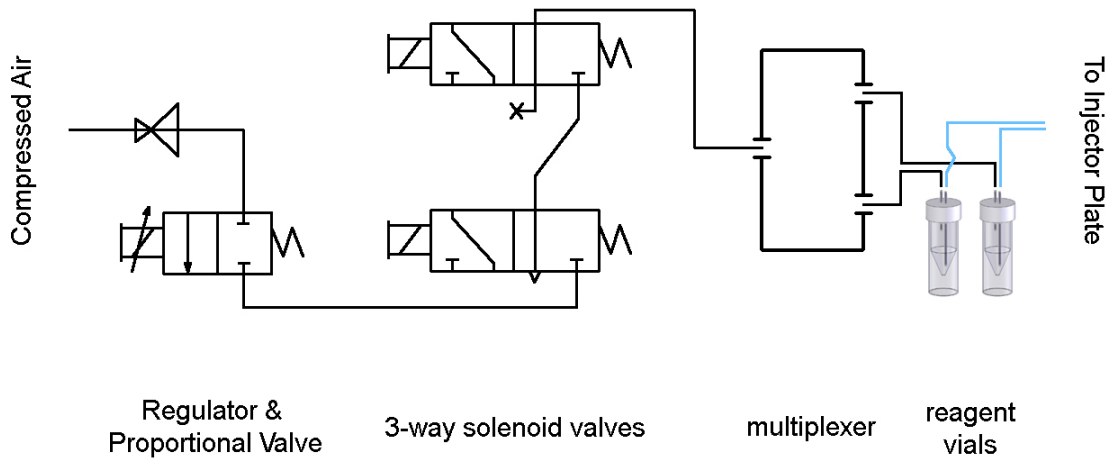
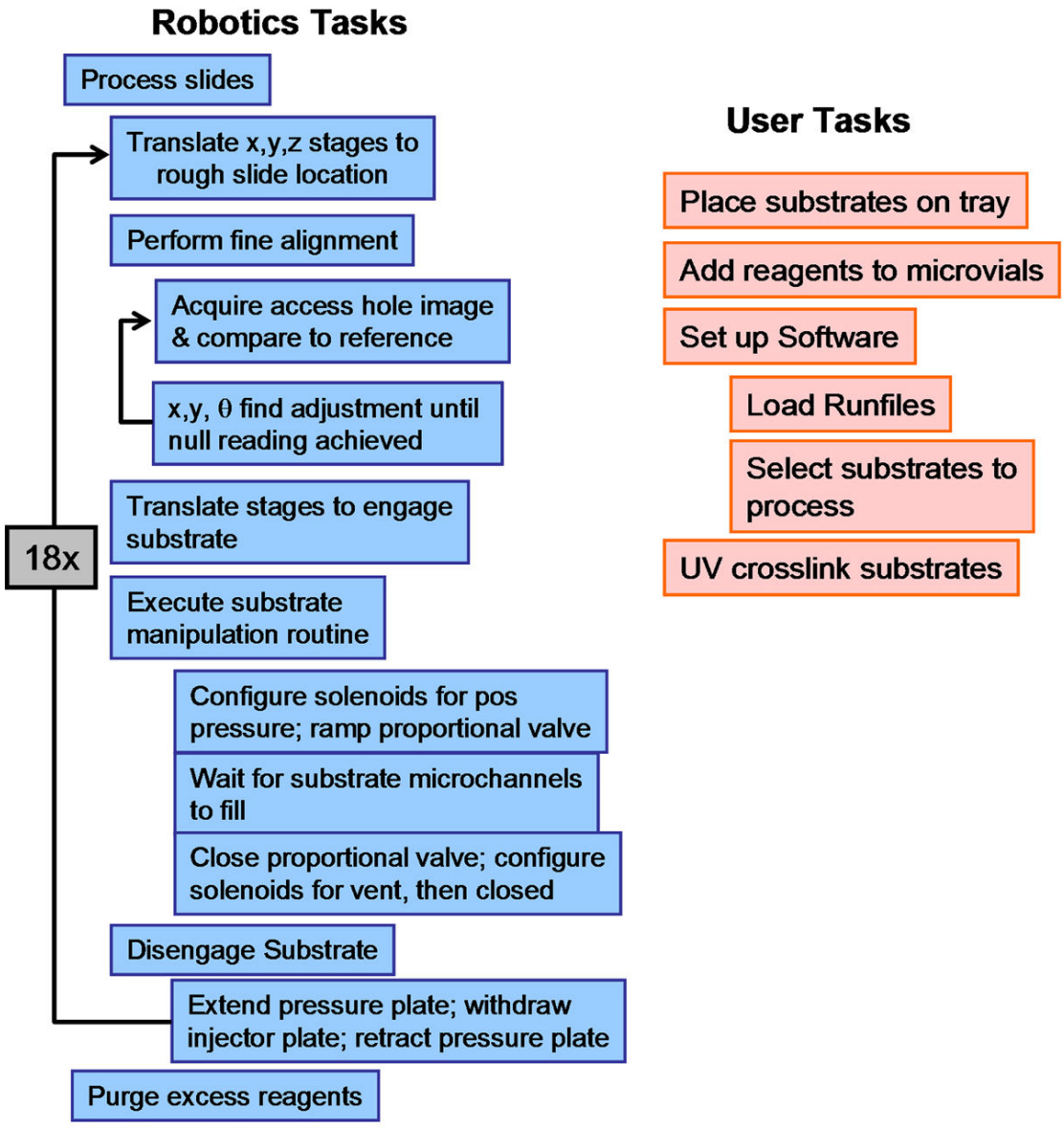


Figure 4.5.5 (A) The distribution of MIF secretion for chambers containing between zero and three cells is plotted. Chambers with more cells exhibit a greater proportion of elevated signals, implying a cumulative effect. Error bars represent standard deviations that are calculated from barcode repeats within individual chambers. **(B)** Heat maps depicting protein secretion for chambers with single cells on each of two chips. Proteins labeled '*' and '**' were contrast enhanced at 10x & 100x original signal levels for clarity. **(C)** Scatter plots of MIF secretion in single cell chambers illustrate the distribution of secretion profiles; the horizontal line represents the average of all the individual measurements.



Scheme 4.5.1 Schematic representation of the instrument's simplified pressure system for driving reagents.



Scheme 4.5.2 Flowchart comparing tasks required of the user and those required of the instrument to prepare a batch of barcode microarray substrates.

4.6 References

1. R. L. Stears, T. Martinsky and M. Schena, *Nature Medicine*, 2003, **9**, 140-145.
2. H. Yue, P. S. Eastman, B. B. Wang, J. Minor, M. H. Doctolero, R. L. Nuttall, R. Stack, J. W. Becker, J. R. Montgomery, M. Vainer and R. Johnston, *Nucleic Acids Research*, 2001, **29**, e41.
3. G. C. Tseng, M.-K. Oh, L. Rohlin, J. C. Liao and W. H. Wong, *Nucleic Acids Research*, 2001, **29**, 2549-2557.
4. D. J. Lockhart, H. Dong, M. C. Byrne, M. T. Follettie, M. V. Gallo, M. S. Chee, M. Mittmann, C. Wang, M. Kobayashi, H. Norton and E. L. Brown, *Nature Biotechnology*, 1996, **14**, 1675 - 1680.
5. S. Singh-Gasson, R. D. Green, Y. Yue, C. Nelson, F. Blattner, M. R. Sussman and F. Cerrina, *Nature Biotechnology*, 1999, **17**, 974-978.
6. L. M. Demers, D. S. Ginger, S.-J. Park, Z. Li, S.-W. Chung and C. A. Mirkin, *Science*, 2002, **296**, 1836 - 1838.
7. F. Huo, Z. Zheng, G. Zheng, L. R. Giam, H. Zhang and C. A. Mirkin, *Science*, 2008, **321**, 1658 - 1660.
8. D. Rose, *Microfluidic technologies and instrumentation for printing DNA microarrays*, Eaton Publishing Co., Natick, MA, 2000.
9. R. S. Kane, S. Takayama, E. Ostuni, D. E. Ingber and G. M. Whitesides, *Biomaterials*, 1999, **20**, 2363-2376.
10. E. Delamarche, A. Bernard, H. Schmid, B. Michel and H. Biebuyck, *Science*, 1997, **276**, 779-781.
11. D. Juncker, H. Schmid, A. Bernard, I. Caelen, B. Michel, N. de Rooij and E. Delamarche, *Journal of Micromechanics and Microengineering*, 2001, **11**, 532-541.
12. C.-E. Ho, C.-C. Chieng, M.-H. Chen and F.-G. Tseng, *Journal of Micromechanics and Microengineering*, 2008, **17**, 309-317.
13. D. A. Chang-Yen, D. G. Myszka and B. K. Gale, *Journal of Microelectromechanical Systems*, 2006, **15**, 1145-1151.
14. C. Thibault, V. Le Berre, S. Casimirius, E. Trévisiol, J. François and C. Vieu, *Journal of Nanobiotechnology*, 2005, **3**.
15. S. A. Lange, V. Benes, D. P. Kern, J. K. H. Hörber and A. Bernard, *Analytical Chemistry*, 2004, **76**, 1641–1647.
16. M. Geissler, E. Roy, J.-S. Deneault, M. Arbour, G. A. Diaz-Quijada, A. Nantel and T. Veres, *Small*, 2009, **5**, 2514–2518.
17. C. M. Niemeyer, T. Sano, C. L. Smith and C. R. Cantor, *Nucleic Acids Research*, 1994, **22**, 5530-5539.
18. R. Wacker, H. Schröder and C. M. Niemeyer, *Analytical Biochemistry*, 2004, **330**, 281-287.
19. R. C. Bailey, G. A. Kwong, C. G. Radu, O. N. Witte and J. R. Heath, *J Am Chem Soc*, 2007, **129**, 1959-1967.

20. E. S. Douglas, R. A. Chandra, C. R. Bertozzi, R. A. Mathies and M. B. Francis, *Lab Chip*, 2007, **7**, 1442-1448.
21. R. Fan, O. Vermesh, A. Srivastava, B. K. Yen, L. Qin, H. Ahmad, G. A. Kwong, C. C. Liu, J. Gould, L. Hood and J. R. Heath, *Nat Biotechnol*, 2008, **26**, 1373-1378.
22. Y. S. Shin, H. Ahmad, Q. Shi, H. Kim, T. A. Pascal, R. Fan, W. A. Goddard III and J. R. Heath, *ChemPhysChem*, 2010, **11**, 3063-3069.
23. J. Liu, C. Hansen and S. R. Quake, *Analytical Chemistry*, 2003, **75**, 4718-4723.
24. A. M. Christensen, D. A. Chang-Yen and B. K. Gale, *Journal of Microelectronics and Microengineering*, 2005, **15**, 928-934.
25. C. L. Hansen, E. Skordalakes, J. M. Berger and S. R. Quake, *PNAS*, 2002, **99**, 16531-16536.
26. C. M. Klapperich, *Expert Rev. Med. Devices*, 2009, **6**, 211-213.
27. H.-Y. Wang, R. L. Malek, A. E. Kwitek, A. S. Greene, T. V. Luu, B. Behbahani, B. Frank, J. Quackenbush and N. H. Lee, *Genome Biology*, 2003, **4**.
28. J. C. Love, J. L. Ronan, G. M. Grotenbreg, A. G. van der Veen and H. L. Ploegh, *Nat Biotech*, 2006, **24**, 703-707.
29. E. M. Bradshaw, S. C. Kent, V. Tripuraneni, T. Orban, H. L. Ploegh, D. A. Hafler and J. C. Love, *Clinical Immunology*, 2008, **129**, 10-18.
30. H. Zhu, G. Stybayeva, J. Silangcruz, J. Yan, E. Ramanculov, S. Dandekar, M. D. George and A. Revzin, *Analytical Chemistry*, 2009, **81**, 8150-8156.
31. C. D. Dumitru, J. D. Ceci, C. Tsatsanis, D. Kontoyiannis, K. Stamatakis, J. H. Lin, C. Patriotis, N. A. Jenkins, N. G. Copeland, G. Kollias and P. N. Tsichlis, *Cell*, 2000, **103**, 1071-1083.
32. A. Aderem and R. J. Ulevitch, *Nature*, 2000, **406**, 782-787.
33. J. Fan and A. B. Malik, *Nature Medicine*, 2003, **9**, 315-321.
34. M. A. Unger, H.-P. Chou, T. Thorsen, A. Scherer and S. R. Quake, *Science*, 2000, **288**, 113 - 116.
35. M. Kærn, T. C. Elston, W. J. Blake and J. J. Collins, *Nature Reviews Genetics*, 2005, **6**, 451-464.
36. N. Rosenfeld, J. W. Young, U. Alon, P. S. Swain and M. B. Elowitz, *Science*, 2005, **307**, 1962 - 1965.
37. P. Openshaw, E. E. Murphy, N. A. Hosken, V. Maino, K. Davis, K. Murphy and A. O'Garra, *Journal of Experimental Medicine*, 1995, **182**, 1357–1367.
38. J.-U. Park, M. Hardy, S. J. Kang, K. Barton, K. Adair, D. k. Mukhopadhyay, C. Y. Lee, M. S. Strano, A. G. Alleyne, J. G. Georgiadis, P. M. Ferreira and J. A. Rogers, *Nature Materials*, 2007, **6**, 782 - 789.
39. I. Barbulovic-Nad, M. Lucente, Y. Sun, M. Zhang, A. R. Wheeler and M. Bussmann, *Critical Reviews in Biotechnology*, 2006, **26**, 237-259.

4.7 Appendix A: Source Code

The following pages contain Labwindows CVI source code used to control and coordinate the robotics hardware. Small pieces of code may be commented out for convenience during development, and as such this should not be considered production-ready code.

```
1   #include   "alignScore.h"
2   #include   <dataskt.h>
3   #include   <analysis.h>
4   #include   <dataacq.h>
5   #include   <easyio.h>
6   #include   <tcpsupp.h>
7   #include   <winerror.h>
8   #include   <formatio.h>
9   #include   <ansi_c.h>
10  #include   <cvirte.h>
11  #include   <utility.h>
12  #include   <userint.h>
13  #include   "lowlvlio.h"
14  #include   "mainPanel.h"
15  #include   "calibration.h"
16
17
18  const int MAXSLIDES = 18;
19  const double MAXPROPV = 10.0;
20  static int Hn_mainPanel;
21  static int Hn_calibratePanel;
22  static int Hn_ICDpanel;
23  static int Hn_SLpanel;
24  static int Hn_FLpanel;
25  static int Hn_AlignPanel;
26  const char sixK_IP[] = "192.168.10.30";
27
28
29
30  DSHandle DataSockets[7] = {0,0,0,0,0,0,0};                    //
    AngleSocket, ColumnSocket, RowSocket, OnlineSocket, JobReadSocket,
    JobWriteSocket, PatternScoreSocket
31
32  int DIOports[8] = {0, 1, 2, 3, 4, 5, -1, -1};                 // [0]
    backlight     [1] psi/vent for pistons [2] low psi/vac        [3]
    psi/vent [4] pressurePlate pistons       [5] contact sensor 2      [6,7]
    undefined
33  int slides[18][5];                        // [0] Cntrl ID      [1]
    activated [2] process complete icon       [3] examine icon [4]
    error icon
34  int CslidePos[18][3];                     // [0] Cntrl ID      [1] Xpos
    label [2] Ypos label
35  int activeSlideColor, inactiveSlideColor, processColor;
36  int pauseFlag;
37  int sixK_TCP = 0;
38  int cutScore = 92;
39
```

```
40  float imagingZ = 0;
41  float slidePos[18][2];                                          // [0] x
    pos     [1] y pos              //indexed to CntrlID of array
    "slides[18][5]" above
42  float CI_xShift, CI_yShift, CI_tShift;
43
44
45  char slidePosFile[80] = "slidePos.dat";
46  char jobsFileName[80] = "jobFiles.jbf";
47
48
49  /****    6k Status bits    ****/
50
51  int activeMotion = 0;
52  float position = 0;
53
54  #define MOTIONCOMPLETE 1
55  #define POSITION 2
56
57  /*************************/
58
59  /*****    6k Variables    *****/
60
61  float encX, encY, encZ, encT;                                   //encoder
    positions (response to TPE);
62
63  /*************************/
64
65
66
67
68
69
70
71
72
73
74
75
76
77  void backlight(int status)
                                                          //turns the
    backlight on or off
78  {
79
80      WriteToDigitalLine (1, "0", 0, 8, 1, status);
81      return;
```

```
82  }
83
84
85
86  void InitializeVars()
87  {
88      slides[0][0] = mainPanel_slide1;
89      slides[1][0] = mainPanel_slide2;
90      slides[2][0] = mainPanel_slide3;
91      slides[3][0] = mainPanel_slide4;
92      slides[4][0] = mainPanel_slide5;
93      slides[5][0] = mainPanel_slide6;
94      slides[6][0] = mainPanel_slide7;
95      slides[7][0] = mainPanel_slide8;
96      slides[8][0] = mainPanel_slide9;
97      slides[9][0] = mainPanel_slide10;
98      slides[10][0] = mainPanel_slide11;
99      slides[11][0] = mainPanel_slide12;
100     slides[12][0] = mainPanel_slide13;
101     slides[13][0] = mainPanel_slide14;
102     slides[14][0] = mainPanel_slide15;
103     slides[15][0] = mainPanel_slide16;
104     slides[16][0] = mainPanel_slide17;
105     slides[17][0] = mainPanel_slide18;
106
107     activeSlideColor = MakeColor (240, 240, 240);
108     inactiveSlideColor = MakeColor (212, 208, 200);
109     processColor = MakeColor (240, 200, 200);
110
111     return;
112 }
113
114
115 void setStatus(char* message)
116 {
117     SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, message);
                  //update status message
118     return;
119 }
120
121
122 void send6kCmd (char* cmd)
123 {
124     char sendCmd[80];
125     char errorMsg[120];
126     char ts[2] = {(char)13, '\0'};
127     int err = 0;
```

```
128
129
130         sprintf(sendCmd, "%s%s\0", cmd, ts);
131
132         err = ClientTCPWrite (sixK_TCP, sendCmd, strlen(sendCmd), 0);
133         if(err < 0)
134         {    sprintf(errorMsg, "TCP error - command: %s was not sent" ,
            sendCmd);
135             MessagePopup("Comm error", errorMsg);
136         }
137
138         return;
139     }
140
141
142     void finishStageMotion()
143     {
144         char sendString[80];
145
146
147         sprintf(sendString, "WAIT(MOV=b0000): WRITE\"#%i\"" ,
            MOTIONCOMPLETE);          //wait while movement on axes; write
148         send6kCmd(sendString);

            //"motion complete" code when stopped.
149
150         while(activeMotion)
151         { //
            Delay(0.5);
                    //causes multiple responses to pile up
152             ProcessSystemEvents();
153         }
154
155         return;
156     }
157
158     void HomeStages()
159     {
160         setStatus("Homing Stages...");
161
162         activeMotion = 1;

            //anticipate upcoming motion
163     //
        send6kCmd("DRIVE1111");
                //enable all stages
164     //
```

```
send6kCmd("HOM111x");
          //intiate homing
165       send6kCmd("4MA1: 4D0: 4GO");
                                                          //set absolute
          mode on axis 4 and go home
166       finishStageMotion();
                                                                  //wait
          until stages finish moving
167       send6kCmd("4MA0");

          //return to incremental mode on axis 4
168
169       setStatus("");
170
171       return;
172   }
173
174
175   void setStageMovement(int setting)              //SETTINGS: 1: all
      stages fast 2: all stages slow, 3: z-stage intermediate
176   {
177       const double vFast = 10.0;
178       const double vMed = 2.0;
179       const double vSlow = 0.75;
180       const double aFast = 50.0;
181       const double aMed = 25.0;
182
183       char cmd[120] = "";
184
185
186       switch(setting)
187       {    case 1: sprintf(cmd, "V %.1f, %.1f, %.1f, %.1f: A %.1f,
          %.1f, %.1f, %.1f", vFast, vFast, vFast, vSlow, aFast, aFast,
          aFast, aMed);
188                   break;
189            case 2: sprintf(cmd, "V %.1f, %.1f, %.1f, %.1f: A %.1f,
               %.1f, %.1f, %.1f", vSlow, vSlow, vSlow, vSlow, aMed, aMed,
               aMed, aMed);
190                   break;
191            case 3: sprintf(cmd, "V %.1f, %.1f, %.1f, %.1f: A %.1f,
               %.1f, %.1f, %.1f", vSlow, vSlow, vMed, vSlow, aMed, aMed,
               aMed, aMed);
192                   break;
193       }
194       send6kCmd(cmd);
195
196       return;
```

```
197  }
198
199
200
201  int readSingleVal(double* xPos, double* yPos, double* tPos, float
     readDelay)
202  {
203       const int maxReads = 30;
                                                                    //maximum
     number of reads before "timeout"
204  // const int cutScore =
     92;                                                          //minimum
     score to accept a pattern **Made Global to allow alteration**
205
206       HRESULT err;
207       int i = 0, j = 0, totalCount = 0;


208       long qString = 0;
209       float pScore = 0;
210       float lxPos = 0, lyPos = 0, ltPos = 0;
211       char instaStatus[50];
212
213
214       DS_Update (DataSockets[0]);
                                                          //update
     angle value
215       DS_Update (DataSockets[1]);
                                                          //        column
216       DS_Update (DataSockets[2]);
                                                          //        row
217       DS_Update (DataSockets[6]);
                                                          //
     pattern score
218
219
                //first check if data is good
220       while(qString != 192 && j++ < 5)
                                                       //192 = "Good"
221            err = DS_GetAttrValue (DataSockets[6], "Quality", CAVT_LONG
               , &qString, sizeof(qString), NULL, NULL);
222       if(j > 5)


223       {    MessagePopup("Comm Error", "Could not retrieve data from
          OPC server!");
224            return 3;
```

```
225         }
226         else


227         {    err = DS_GetDataValue (DataSockets[6], CAVT_FLOAT, &pScore,
             sizeof(pScore), NULL, NULL);            //read in pattern quality
228             if(err < 0) CA_DisplayErrorInfo(DataSockets[6],
                "DataSocket Error", err, NULL);
229         }
230         sprintf(instaStatus, "Analyzing image... %i%% match" , (int)
            pScore);          //update quality on status bar
231         setStatus(instaStatus);
232
233
234         while((pScore < cutScore) && (++totalCount < maxReads))
                                  //while pattern quality is substandard
235         {    Delay(readDelay);
                                                                        //
            wait for new reading
236             DS_Update (DataSockets[0]);
                                                                //update
                angle value
237             DS_Update (DataSockets[1]);
                                                                //        column
238             DS_Update (DataSockets[2]);
                                                                //        row
239             DS_Update (DataSockets[6]);
                                                                //
                pattern score
240
241             err = DS_GetDataValue (DataSockets[6], CAVT_FLOAT, &pScore,
                 sizeof(pScore), NULL, NULL);            // read new value
242             if(err < 0) CA_DisplayErrorInfo(DataSockets[6],
                "DataSocket Error", err, NULL);
243             sprintf(instaStatus, "Analyzing image... %i%% match" , (int
                )pScore);
244             setStatus(instaStatus);
245         }
246         if(totalCount >= maxReads) return 2;
                                                    //on timeout, return
            error
247 248
249         err = DS_GetDataValue (DataSockets[1], CAVT_FLOAT, &lyPos,
            sizeof(pScore), NULL, NULL);                    // and write to
            panel
250         if(err < 0) CA_DisplayErrorInfo(DataSockets[1], "DataSocket
```

```
        Error", err, NULL);
251
252     err = DS_GetDataValue (DataSockets[2], CAVT_FLOAT, &lxPos,
        sizeof(pScore), NULL, NULL);
253     if(err < 0) CA_DisplayErrorInfo(DataSockets[2], "DataSocket
        Error", err, NULL);
254
255     err = DS_GetDataValue (DataSockets[0], CAVT_FLOAT, &ltPos,
        sizeof(pScore), NULL, NULL);
256     if(err < 0) CA_DisplayErrorInfo(DataSockets[0], "DataSocket
        Error", err, NULL);
257
258
259     *xPos = (double)lxPos;
260     *yPos = (double)lyPos;
261     *tPos = (double)ltPos;
262
263     return 0;
264
265 /******* RETURN VALS ********
266  0 = great success!
267  1 = not used in this function
268  2 = unable to match pattern
269  3 = OPC server errors
270 ***************************/
271 }
272
273
274 int readCameraVal(double* xAvg, double* yAvg, double* tAvg)
275 {
276     #define avgBlock
        5
        //number of reads to average per value
277     #define maxReads
        5
        //maximum number of reads before "timeout"
278     const float readDelay = 1.0;
                                                                //delay
        between camera update requests
279     const float xyMaxStdDev = 0.030;
                                                              //maximum
        allowable standard deviation
280     const float tMaxStdDev = 0.1;
281
282     int status, i, j, totalCount;
283     double xArr[avgBlock * maxReads],
284            yArr[avgBlock * maxReads],
```

```
285             tArr[avgBlock * maxReads];
                                                                    //arrays
                for std deviation calc
286         int arrSize = avgBlock * maxReads;
                                                                //calculate for
            convenience later
287         double xMean, yMean, tMean;

            //calculated array mean values
288         double xSD = 100, ySD = 100, tSD = 100;
                                                            //calculated array
            standard deviation
289
290
291         setStatus("Idle - allowing imaging system to settle" );
292         backlight(1);

                //turn on backlight
293         Delay(3);

                    //allow settle/stabilize
294         setStatus("Analyzing image...");
295
296
297         totalCount = 0;
298         while(((xSD > xyMaxStdDev) || (ySD > xyMaxStdDev) || (tSD >
            tMaxStdDev)) && (totalCount < maxReads))
299         {   j = totalCount * avgBlock;
300             for(i = j; i < j + avgBlock; i++)
                                                                //get values to
                compute quality factor
301             {   status = readSingleVal(&xArr[i], &yArr[i], &tArr[i],
                readDelay);          // read values
302                 if(status)

                            // if error thrown
303                 {   backlight(0);

                    //    turn off backlight
304                     return status;

                            //    exit function immediately
305                 }
306                 Delay(readDelay);

                    //  wait for updated value
307             }
```

```
308
309             Mean (xArr, i, &xMean);

                //calculate mean values for x,y,t arrays
310             Mean (yArr, i, &yMean);

311             Mean (tArr, i, &tMean);
312
313             StdDev (xArr, i, &xMean, &xSD);
                                                        //calculate
                standard deviations for arrays
314             StdDev (yArr, i, &yMean, &ySD); 315
                StdDev (tArr, i, &tMean, &tSD);
316
317             totalCount++;
318         }
319
320         backlight(0);
321         if(totalCount >= maxReads) return 1;
                                                    //if std deviation
          "timeout" return err
322
323
324         Mean (xArr, i, xAvg);

          //take average of collected values and
325         Mean (yArr, i, yAvg);

          //place in referenced values
326         Mean (tArr, i, tAvg);
327
328         return 0;
329
330     /******* RETURN VALS ********
331      0 = great success!
332      1 = unstable pattern score
333      2 = unable to match pattern
334      3 = OPC server errors
335     ***************************/
336     }
337
338
339
340
341     int PopulateRings()
342     {
343         int i = 0, j = 0;
```

```
344         int fileSize = 0;
345         char entryName[80], opcEntry[80];
346         FILE* jobFiles;
347         long qString;
348         HRESULT err;
349
350
351
352
                        //read current job file from OPC server
353         while(qString != 192 && j++ < 5)
                                                                //192 = "Good"
354             err = DS_GetAttrValue (DataSockets[4], "Quality", CAVT_LONG
                , &qString, sizeof(qString), NULL, NULL);         //check
                connection
355         if(j > 5) MessagePopup("Comm Error", "Could not retrieve data
            from OPC server!");
356         else
357         {   err = DS_GetDataValue (DataSockets[4], CAVT_CSTRING,
            opcEntry, sizeof(opcEntry), NULL, NULL);                //retrieve
            value
358             if(err < 0) CA_DisplayErrorInfo(DataSockets[4],
                "DataSocket Error", err, NULL);
359         }
360
361
362
363         if(!(GetFileInfo (jobsFileName, &fileSize)))
                                                //if file doesn't exist
364         {   sprintf(entryName, "There was a problem opening '%s'.
            Please \nensure that it is in this program's root directory" ,
            jobsFileName);
365             MessagePopup ("Error", entryName);
366            return 1;
367         }
368
369         jobFiles = fopen (jobsFileName, "r");
370         while(!feof (jobFiles))

                //while end of file not reached
371         {   //ReadFile (jobFiles, entryName,
            sizeof(entryName)/sizeof(entryName[0]));                // read
            entry
372             fgets (entryName, sizeof(entryName), jobFiles);
373             if(entryName[strlen(entryName)-1] == '\n') entryName[strlen
                (entryName)-1] = '\0';      // if CR present,strip it
374             if(strcmp(entryName, opcEntry) == 0) j = i;
```

```
                                                        // if entry
            matches current OPC value, record position
375         InsertListItem (Hn_mainPanel, mainPanel_CognexRing, -1,
            entryName, i++);                // add it to main panel
            Cognex ring
376     }
377
378     InsertListItem (Hn_mainPanel, mainPanel_CognexRing, -1, "new
        job file", -1);                //add instruction entry for new job
        names
379     SetCtrlVal (Hn_mainPanel, mainPanel_CognexRing, j);
                                                    //set value to match
        current job on OPC server
380 381
382     fflush (jobFiles); 383
        fclose (jobFiles);

        //flush & close files
384 385
386     return 0;
387 }
388
389
390 int loadSlidePos()
391 {
392     int i;
393     char inString[100] = "";
394     float xVal, yVal;
395     FILE* SLfile;
396     long fileSize;
397
398
399
400     if(!(GetFileInfo (slidePosFile, &fileSize)))
                                            //if file doesn't exist
401     { sprintf(inString, "There was a problem opening '%s'!" ,
        slidePosFile);
402         MessagePopup ("Error", inString);
403         return 1;
404     }
405
406
407     i = 0;
408     SLfile = fopen (slidePosFile, "r");
409     while((!feof (SLfile)) && (i < MAXSLIDES))
```

```
                                                    //while end of file not
        reached
410     {   fgets (inString, sizeof(inString), SLfile);
                                                // get data
411         sscanf (inString, "%f, %f\n", &xVal, &yVal);
                                                    // extract info
412         slidePos[i][0] = xVal;
                                                                        // and
            update in master array
413         slidePos[i][1] = yVal;
414         i++;
415     }
416
417     fflush (SLfile); 418
        fclose (SLfile);

        //flush & close file
419
420     return 0;
421 }
422
423 424
425 int writeConfig()
426 {
427     char writeString[80];
428     int fileSize = 0;
429     char fileName[] = "last_used.cfg";
430     FILE* configFile;
431
432
433     SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, "Writing
        configuration file...");       //update status
434     ProcessSystemEvents();
435
436     configFile = fopen (fileName, "w");
437
438     fprintf(configFile,    "**FILES**\n");
439     fprintf(configFile,    "jobs file: %s\n", jobsFileName);
440     fprintf(configFile,    "slides file: %s\n", slidePosFile);
441     fprintf(configFile,    "\n");
442     fprintf(configFile,    "**PROGRAM VARIABLES**\n");
443     fprintf(configFile,    "ICD: %f %f %f\n", CI_xShift, CI_yShift,
        CI_tShift);
444     fprintf(configFile,    "FL: %f\n", imagingZ);
445
446
```

```
447         fflush(configFile);
448         fclose(configFile);
449
450         Delay(0.5);
451         SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, "");
                                        //update status
452         ProcessSystemEvents();
453
454         return 0;
455     }
456
457 458
459     int readConfig()
460     {
461         int fileSize = 0;
462         char fileName[] = "last_used.cfg";
463         char readString[80];
464         char stringVar[80];
465         FILE* configFile;
466
467
468
469         SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, "Reading
            program options...");           //update status
470         ProcessSystemEvents();
471
472         if(!(GetFileInfo (fileName, &fileSize)))
                                                  //if file doesn't exist
473         {    sprintf(readString, "There was a problem opening '%s'.
            Please \nensure that it is in this program's root directory" ,
            fileName);
474             MessagePopup ("Error", readString);
475            return 1;
476         }
477
478         configFile = fopen (fileName, "r");
479
480         fgets (readString, sizeof(readString), configFile);
                                        //read **FILES** header
481         fgets (readString, sizeof(readString), configFile);
482             sscanf(readString, "jobs file: %s\n", stringVar);
                                          //process jobs file
483             strcpy(jobsFileName, stringVar);


484         fgets (readString, sizeof(readString), configFile);
```

```
485             sscanf(readString, "slides file: %s\n", stringVar);
                                        //process slides file
486             strcpy(slidePosFile, stringVar);
487         fgets (readString, sizeof(readString), configFile);
                                        //swallow space
488         fgets (readString, sizeof(readString), configFile);
                                        //read **PROGRAM VARIABLES** header
489         fgets (readString, sizeof(readString), configFile);
490             sscanf(readString, "ICD: %f %f %f\n", &CI_xShift, &
                CI_yShift, &CI_tShift);          //get camera-injector shifts
491         fgets (readString, sizeof(readString), configFile);
492             sscanf(readString, "FL: %f\n", &imagingZ);
                                                  //get focal length
493 494 495
496         if(PopulateRings()) return 1; 497
            if(loadSlidePos()) return 1;
498
499
500         Delay(0.5);
501         SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, "");
                                        //update status
502         ProcessSystemEvents();
503
504
505         return 0;
506     }
507
508     int read_TCPdata()
509     {
510         char msg[200] = "";
511         int err = 0;
512         int errFlag = 0;
513     // char stdResponse[5] = {(char)13, (char)10, '>', ' ',
        '\0'};                      //standard response from 6k
514         char stdResponse[5] = {(char)13, (char)10, '>', '\0'};
515         char* errResponse = "?";
                                                                    //found in
            error response from 6k
516     // char* errFound = NULL;
517         char* newString = NULL;
518         char* subString = NULL;
519
520
521         err = ClientTCPRead (sixK_TCP, msg, sizeof(msg), 0);
                                                  //read TCP data
```

```
522         if(err<0) MessagePopup ("Comm error", "Error reading response
            from 6k controller");        // check for read errors
523
524 525
526         newString = strtok(msg, stdResponse);
                                                        //tokenize response
527         while(newString)

            //while token found
528         {    if(strcmp(newString, " ") == 0)
                                                          //   blank space?
529              {    }

                       //     do nothing
530
531              else if(strpbrk(newString, errResponse))
                                                    // error response?
532              {    errFlag = 1; }

                 //     flag query submission
533
534              else if(subString = strpbrk(newString, "#"))
                                                 // predefined response?
535              {    sscanf(subString, "#%i", &err);
                                                          //     read code
536                   switch(err)

                       //     process accordingly
537                   {    case MOTIONCOMPLETE:
538                             activeMotion = 0;
539                             break;
540                   }
541              }
542
543              else if(strstr(newString, "*TPE"))
                                                        // position string?
544              {    sscanf(msg, "*TPE%f,%f,%f,%f", &encX, &encY, &encZ, &
                 encT);            //     record values
545    setStatus(msg);       }
546
547              else
548              {        MessagePopup ("6k Response", newString);        }
                                    // else show message
549 550
551              newString = strtok(NULL, stdResponse);
```

```
                                                    //   grab next token
552         }
553
554         if(errFlag) send6kCmd("TCMDER");
                                                                //if error
            generated, request offending command
555 556
557         return 0;
558     }
559
560 561
562     int TCP_callback(unsigned handle, int xType, int errCode, void *
        callbackData)
563     {
564         char errorMsg[80] = "";
565
566
567         if(xType == TCP_DISCONNECT)
568         {    sprintf(errorMsg, "TCP connection to the 6k controller was
            lost! Error code: %i", &errCode);
569             MessagePopup ("TCP/IP Error", errorMsg);
570         }
571         else
572         {    read_TCPdata();
573         }
574
575         return 0;
576      }
577
578
579     int ConnectOPCserver()
580     {
581         HRESULT err;
582         char url[6][80];
583         DSEnum_Status status;
584         char error[80];
585
586         SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, "Connecting to
            OPC servers...");        //update status
587         ProcessSystemEvents();
588
                    //copy opc server addresses into array
589         sprintf(url[0], "opc://localhost/Cognex In-Sight OPC
            Server/IS5400R.AngleShift");
590         sprintf(url[1], "opc://localhost/Cognex In-Sight OPC
```

```
        Server/IS5400R.ColumnShift");
591     sprintf(url[2], "opc://localhost/Cognex      In-Sight OPC
        Server/IS5400R.RowShift");
592     sprintf(url[3], "opc://localhost/Cognex      In-Sight OPC
        Server/IS5400R.Online");
593     sprintf(url[4], "opc://localhost/Cognex      In-Sight OPC
        Server/IS5400R.JobName");
594     sprintf(url[5], "opc://localhost/Cognex      In-Sight OPC
        Server/IS5400R.PatternScore");
595
596
597     err = DS_Open (url[0], DSConst_Read, NULL, NULL, &DataSockets[0
        ]);             //open server address for read access
598     if(err < 0)

             //if error
599     {   MessagePopup ("Comm Error", "Error connecting to Angle
        server!");          // indicate which address caused error
600         CA_DisplayErrorInfo(DataSockets[0], "DataSocket Error", err
            , NULL);        // and show error
601         return 1;
602     }
603
604     err = DS_Open (url[1], DSConst_Read, NULL, NULL, &DataSockets[1
        ]);
605     if(err < 0)
606     {   MessagePopup ("Comm Error", "Error connecting to Column
        server!");
607         CA_DisplayErrorInfo(DataSockets[1], "DataSocket Error", err
            , NULL);
608         return 1;
609     }
610
611     err = DS_Open (url[2], DSConst_Read, NULL, NULL, &DataSockets[2
        ]);
612     if(err < 0)
613     {   MessagePopup ("Comm Error", "Error connecting to Row
        server!");
614         CA_DisplayErrorInfo(DataSockets[2], "DataSocket Error", err
            , NULL);
615         return 1;
616     }
617
618     err = DS_Open (url[3], DSConst_Read, NULL, NULL, &DataSockets[3
        ]);
619     if(err < 0)
620     {   MessagePopup ("Comm Error", "Error connecting to Online
```

```
        server!");
621         CA_DisplayErrorInfo(DataSockets[3], "DataSocket Error", err
            , NULL);
622         return 1;
623     }
624
625     err = DS_Open (url[4], DSConst_Read, NULL, NULL, &DataSockets[4
        ]);
626     if(err < 0)
627     {   MessagePopup ("Comm Error", "Error connecting to JobRead
        server!");
628         CA_DisplayErrorInfo(DataSockets[4], "DataSocket Error", err
            , NULL);
629         return 1;
630     }
631
632     err = DS_Open (url[4], DSConst_Write, NULL, NULL, &DataSockets[
        5]);
633     if(err < 0)
634     {   MessagePopup ("Comm Error", "Error connecting to JobWrite
        server!");
635         CA_DisplayErrorInfo(DataSockets[5], "DataSocket Error", err
            , NULL);
636         return 1;
637     }
638
639     err = DS_Open (url[5], DSConst_Read, NULL, NULL, &DataSockets[6
        ]);
640     if(err < 0)
641     {   MessagePopup ("Comm Error", "Error connecting to
        PatternScore server!");
642         CA_DisplayErrorInfo(DataSockets[5], "DataSocket Error", err
            , NULL);
643         return 1;
644     }
645
646
647     SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, "Connecting to
        OPC servers... Connected!");
648     Delay(0.2);
649     ProcessSystemEvents();
650
651     setStatus("Connecting to 6k controller..." );
652     err = 1;
653     while(err)
654     {   err = ConnectToTCPServer (&sixK_TCP, 5002, sixK_IP,
        TCP_callback, NULL, 0);        //connect to 6k TCP server
```

```
655             if(err)

                                    //if error occurred
656             {   sprintf(error, "Error connecting to 6k controller:
                %s\n\n Retry connection?\n", GetTCPErrorString (err));
                // give msg, request response
657                 err = ConfirmPopup ("Comm Error", error);
                                                        // 1 = retry, 0 =
                    fail
658                 setStatus("Error connecting to 6k controller!" );
659                 if(err == 0) return 1;

                    //   if fail, exit function
660             }
661         }
662         setStatus("Connecting to 6k controller... Connected!" );
663         Delay(0.1);
664         ProcessSystemEvents();
665
666         return 0;
667     }
668
669
670     int main (int argc, char *argv[])
671     {
672         if (InitCVIRTE (0, argv, 0) == 0)
673             return -1; /* out of memory */
674         if ((Hn_mainPanel = LoadPanel (0, "mainPanel.uir", mainPanel))
            < 0)
675             return -1;
676
677
678
679         DisplayPanel (Hn_mainPanel);
680
681         InitializeVars();
682         if(ConnectOPCserver())
683         {   SetCtrlAttribute (Hn_mainPanel, mainPanel_cmdStartRun,
            ATTR_DIMMED, 1);
684             SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, "There were
                errors connecting to the Cognex OPC server" );
685         }
686         if(readConfig())
687         {   SetCtrlAttribute (Hn_mainPanel, mainPanel_cmdStartRun,
            ATTR_DIMMED, 1);
688             SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, "There were
                errors initializing the program" );
```

```
689         }
690
691         RunUserInterface ();
692         DiscardPanel (Hn_mainPanel);
693         return 0;
694     }
695
696 697
698     int CVICALLBACK click_slide (int panel, int control, int event,
699             void *callbackData, int eventData1, int eventData2)
700     {
701         int i=0;
702         char message[80];
703
704
705         if(event != EVENT_LEFT_CLICK) return 0;
706
707
708         while(control != slides[i++][0]);

            //find which slide was clicked
709         if(slides[--i][1])

                        //if slide is activated
710         {   slides[i][1] = 0;

                    // deactivate slide
711             SetCtrlAttribute (Hn_mainPanel, slides[i][0],
                ATTR_FRAME_COLOR, inactiveSlideColor);        // remove
                indicator
712         }
713         else
714         {   slides[i][1] = 1;

                    // else activate slide
715             SetCtrlAttribute (Hn_mainPanel, slides[i][0],
                ATTR_FRAME_COLOR, activeSlideColor);          // add indicator
716         }
717
718
719         return 0;
720     }
721
722 723 724
```

```
725
726   int CVICALLBACK SelectAll (int panel, int control, int event,
727            void *callbackData, int eventData1, int eventData2)
728   {
729       int i;
730
731       if(event != EVENT_COMMIT) return 0;
732
733       for(i = 0; i < MAXSLIDES; i++)
          //go through each slide
734       {    slides[i][1] = 1;
                   // select slide
735            SetCtrlAttribute (Hn_mainPanel, slides[i][0],
               ATTR_FRAME_COLOR, activeSlideColor);          // color
               appropriately
736       }
737
738       return 0;
739   }
740
741 742
743   int CVICALLBACK SelectNone (int panel, int control, int event,
744            void *callbackData, int eventData1, int eventData2)
745   {
746       int i;
747
748       if(event != EVENT_COMMIT) return 0;
749
750       for(i = 0; i < MAXSLIDES; i++)
          //go through each slide
751       {    slides[i][1] = 0;
                   // deselect slide
752            SetCtrlAttribute (Hn_mainPanel, slides[i][0],
               ATTR_FRAME_COLOR, inactiveSlideColor);        // color
               appropriately
753       }
754
755       return 0;
756   }
757
758
```

```
759
760  int updateStatus(int slideNo, int statusCode)
761  {
762       int slideX, slideY, slideWidth, slideHeight;
763       int iconX, iconY;
764       int PicID;
765
766
767       if(!statusCode)

                          //if start processing
768       {    SetCtrlAttribute (Hn_mainPanel, slides[slideNo][0],
          ATTR_FRAME_COLOR, processColor);        // color appropriately
769            return 0;

                                    //   exit function
770       }
771
772       SetCtrlAttribute (Hn_mainPanel, slides[slideNo][0],
          ATTR_FRAME_COLOR, activeSlideColor);        //return to normal color
773       GetCtrlAttribute (Hn_mainPanel, slides[slideNo][0], ATTR_LEFT,
          &slideX);                          //get slide position attributes
774       GetCtrlAttribute (Hn_mainPanel, slides[slideNo][0], ATTR_TOP, &
          slideY);
775       GetCtrlAttribute (Hn_mainPanel, slides[slideNo][0], ATTR_WIDTH,
           &slideWidth);
776       GetCtrlAttribute (Hn_mainPanel, slides[slideNo][0], ATTR_HEIGHT
          , &slideHeight);
777
778       iconX = slideX + (int)((slideWidth - 45)/2);
                                                                  //determine
          icon placement
779       iconY = slideY + (int)((slideHeight - 45)/2);
780       slides[slideNo][statusCode] = NewCtrl (Hn_mainPanel,
          CTRL_PICTURE, "", iconY, iconX);                    //create
          picture control
781       SetCtrlAttribute (Hn_mainPanel, slides[slideNo][statusCode],
          ATTR_FRAME_VISIBLE, 0);                    //turn off frame
782       SetCtrlAttribute (Hn_mainPanel, slides[slideNo][statusCode],
          ATTR_WIDTH, 40);                           //set width (thinner to
          elim fringing bug)
783       SetCtrlAttribute (Hn_mainPanel, slides[slideNo][statusCode],
          ATTR_HEIGHT, 45);                          //set height
784       SetCtrlAttribute (Hn_mainPanel, slides[slideNo][statusCode],
          ATTR_FIT_MODE, VAL_PICT_CENTER);           //center image in box
785       SetCtrlAttribute (Hn_mainPanel, slides[slideNo][statusCode],
          ATTR_PICT_BGCOLOR, activeSlideColor); //set background color
```

```
786
787
788         switch(statusCode)

                                //switch case
789         {    case 2:

                                        // case: finished
790                 GetBitmapFromFile ("imgs/checkmark.pcx", &PicID);
                                                                //     load
                    checkmark file
791                 SetCtrlBitmap (Hn_mainPanel, slides[slideNo][statusCode
                    ], 0, PicID);                               //     set in
                    picture frame
792                 break;
793
794             case 3:

                                            // case: usr examine
795                 GetBitmapFromFile ("imgs/mg.pcx", &PicID);

                    //     load magnifying glass file
796                 SetCtrlBitmap (Hn_mainPanel, slides[slideNo][statusCode
                    ], 0, PicID);                               //     set in
                    picture frame
797                 iconY = slideY + (slideHeight - 50);

                    //     shift to bottom of slide
798                 SetCtrlAttribute (Hn_mainPanel, slides[slideNo][
                    statusCode], ATTR_TOP, iconY);
799                 break;
800
801             case 4:

                                            // case: error
802                 GetBitmapFromFile ("imgs/error.pcx", &PicID);
                                                                    //
                    load error file
803                 SetCtrlBitmap (Hn_mainPanel, slides[slideNo][statusCode
                    ], 0, PicID);                               //     set in
                    picture frame
804                 iconY = slideY + (slideHeight - 50);

                    //     shift to bottom of slide
805                 SetCtrlAttribute (Hn_mainPanel, slides[slideNo][
                    statusCode], ATTR_TOP, iconY);
806                 break;
```

```
807         }
808
809         return 0;
810     }
811
812 813
814     int CVICALLBACK updateProgressBar (int panel, int control, int
        event,
815             void *callbackData, int eventData1, int eventData2)
816     {
817         double currVal, tmrInterval, maxVal;

            //for each time teh buzzer rings
818
819         GetCtrlVal (Hn_mainPanel, mainPanel_progressSlide, &currVal);
                                                    //get current state,
            max value, and interval
820         GetCtrlAttribute (Hn_mainPanel, mainPanel_tmrProgressBar,
            ATTR_INTERVAL, &tmrInterval);
821         GetCtrlAttribute (Hn_mainPanel, mainPanel_progressSlide,
            ATTR_MAX_VALUE, &maxVal);
822
823
824         if((currVal + tmrInterval) >=      maxVal)

            //if we've reached the end
825         {    SetCtrlVal (Hn_mainPanel, mainPanel_progressSlide, maxVal);
                                                // set status bar to 100%
826             Delay(1);

                                               // pause 1 second for effect
827             SetCtrlAttribute (Hn_mainPanel, mainPanel_tmrProgressBar,
                ATTR_ENABLED, 0);                          // turn off timer
828             SetCtrlAttribute (Hn_mainPanel, mainPanel_progressSlide,
                ATTR_VISIBLE, 0);                           // hide progress
                slider
829         }
830         else

                                               //otherwise
831             SetCtrlVal (Hn_mainPanel, mainPanel_progressSlide, currVal
                + tmrInterval);                           // update with new
                value
832 833
834         return 0;
```

```
835  }
836
837
838  int setProgressBar (double maxVal, double interval)
839  {
840      SetCtrlAttribute (Hn_mainPanel, mainPanel_tmrProgressBar,
         ATTR_INTERVAL, interval);
841      SetCtrlAttribute (Hn_mainPanel, mainPanel_progressSlide,
         ATTR_MAX_VALUE, maxVal);
842      SetCtrlVal (Hn_mainPanel, mainPanel_progressSlide, 0.0);
843      return 0;
844  }
845
846  void startProgressBar()
847  {
848      SetCtrlAttribute (Hn_mainPanel, mainPanel_tmrProgressBar,
         ATTR_ENABLED, 1);
849      SetCtrlAttribute (Hn_mainPanel, mainPanel_progressSlide,
         ATTR_VISIBLE, 1);
850
851      return;
852  }
853
854
855
856  int rampVoltage(int channel, float newVoltage, float duration)
857  {
858      static double currVoltage[2] = {0,0};
859
860      double waveForm[1000];
861      char txtChannel[10];
862      double updatePerSec;
863      double voltageIncrement;
864      int i;
865      char valveLabel[2][10] = {"solution", "piston"};
866      char statusMessage[100], priorMessage[100];
867
868
869
870
871      GetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, priorMessage);
872
873
874      sprintf(statusMessage, "Ramping %s pressure over %.1f seconds" ,
          valveLabel[channel], duration);
875      sprintf(txtChannel, "%i", channel);
                                                  //copy channel info
```

```
        into string
876     voltageIncrement = (newVoltage - currVoltage[channel])/1000;
                        //determine increment
877     if(voltageIncrement == 0) return 0;
                                                    //if no change, exit
878     waveForm[0] = currVoltage[channel];
879     for(i = 1; i < 1000; i++)
                                                                //fill
        voltage array accordingly
880         waveForm[i] = waveForm[i-1] + voltageIncrement;
881     waveForm[999] = newVoltage;
882
883     updatePerSec = 1000/duration;
884     setStatus(statusMessage);
885     AOGenerateWaveforms (1, txtChannel, updatePerSec, 1000, 1,
        waveForm, &i);
886     setStatus(priorMessage);
887
888     currVoltage[channel] = newVoltage;
889
890     return 0;
891 }
892
893 894
895 int CVICALLBACK expungeSolutions (int panel, int control, int event,
896         void *callbackData, int eventData1, int eventData2)
897 {
898     char cmd[80];
899     double startTime;
900     double expungeLength = 30.0;
901
902     if(event != EVENT_COMMIT) return 0;
903
904
905     setStatus("Expunging left-over solutions" );
906
907     setStageMovement(1);

        //enable fast motion
908     send6kCmd("MA1111");
                                                                //set
        absolute positioning mode on all axes
909         activeMotion = 1;

            //anticipate upcoming motion
910         sprintf(cmd, "D 25.0, 190.0, 10.0, 0: @GO" );
```

```
                                        //construct positioning string
911             send6kCmd(cmd);

                //and write to 6k
912             finishStageMotion();
                                                                    //wait
                until stages finish moving
913
914             activeMotion = 1;

                //anticipate upcoming motion
915             sprintf(cmd, "D 25.0, 190.0, 60.0, 0: @GO" );
                                        //construct positioning string
916             send6kCmd(cmd);

                //and write to 6k
917             finishStageMotion();
                                                                    //wait
                until stages finish moving
918         send6kCmd("@MA0");

            //restore relative positioning
919
920
921         setProgressBar (expungeLength, 5);
922         WriteToDigitalLine (1, "0", DIOports[2], 8, 0, 1);
                                    //toggle valve to pressurize DNA
            solutions
923         WriteToDigitalLine (1, "0", DIOports[3], 8, 0, 1);
924         rampVoltage(0, MAXPROPV, 10);
                                                        //open prop
            valve to 100%
925         startTime = Timer ();
                                                                //note
            start time
926         startProgressBar();
                                                                  //and
            start progress bar timer
927
928
929         while((Timer() - startTime) < expungeLength)
                                        //wait for user-specified
            duration
930         ProcessSystemEvents();
931
932
933         WriteToDigitalLine (1, "0", DIOports[3], 8, 0, 0);
```

```
                                        //switch from pressure to vent
934         rampVoltage(0, 0.0, 10);
                                                                    //then
            close pressure valves
935         WriteToDigitalLine (1, "0", DIOports[2], 8, 0, 0);
936
937
938
939         send6kCmd("MA1111");
                                                                    //set
            absolute positioning mode on all axes
940             activeMotion = 1;

                //anticipate upcoming motion
941             sprintf(cmd, "D 35.0, 0.0, -40.0, 0: @GO");
                                            //construct positioning string
942             send6kCmd(cmd);

                //and write to 6k
943             finishStageMotion();
                                                                    //wait
                until stages finish moving
944         send6kCmd("@MA0");
945
946         setStatus("");
947
948         return 0;
949     }
950
951
952     int PositionInjector(int slideNo)
953     {
954         const double xyTolerance = 0.025;
955
956         char cmd[80] = "";
957         double xPos = .026, yPos = .026, tPos = 0;
958         int posStatus = 0, secondChance = 1;
959
960
961
962         setStageMovement(1);
                                                                    //move
            to slide imaging position
963         send6kCmd("MA1111");
                                                                    //set
            absolute positioning mode on all axes
964             activeMotion = 1;
```

```
            //anticipate upcoming motion
965         sprintf(cmd, "D %f, %f, %f, 0: @GO", slidePos[slideNo][0],
            slidePos[slideNo][1], imagingZ);          //construct
            positioning string
966         send6kCmd(cmd);

            //and write to 6k
967         finishStageMotion();
                                                                  //wait
            until stages finish moving
968       send6kCmd("@MA0");

          //restore relative positioning
969
970
971       posStatus = readCameraVal(&xPos, &yPos, &tPos);
                                      //get offset values
972       while((fabs(xPos) > xyTolerance) || (fabs(yPos) > xyTolerance))
                    //while out of spec
973       {
974           if(posStatus)

            //if error generated, give message
975           {    if(secondChance)
                                                                  //
            check for "second chance" ***see below***
976               {    secondChance--;
                                                                   //
                reduce chance
977                 setStatus("Performing \"second chance\" adjustment"
                    );
978                 Delay(2);
979                 xPos = 0.075;

                    // set minor shift for x & y coordinates
980                 yPos = 0.075;
981               }
982               else
983               {    if(posStatus == 1) setStatus("Unable to achieve
                  stable alignment reading");         //MessagePopup
                  ("Alignment Error", "Unable to achieve stable
                  alignment reading");
984                 if(posStatus == 2) setStatus("Unable to match
                    pattern");                              //MessagePopup
                    ("Alignment Error", "Unable to match pattern");
985                 if(posStatus == 3) setStatus("Error communicating
```

```
                    with vision OPC server");        //MessagePopup
                    ("Alignment Error", "Error communicating with
                    vision OPC server");
986
987                 updateStatus(slideNo, 4);
                                                              //add error
                    icon
988 //              updateStatus(slideNo,
    0);                                          //recover "active" color
989                 return 1;
990             }
991         }
992         setStatus("Performing fine adjustments to injector
            position");
993
994         setStageMovement(2);
                                                                  //set
            slow motion for better precision
995         activeMotion = 1;

            //anticipate upcoming motion
996         sprintf(cmd, "D %f, %f, 0, 0: @GO", xPos, (yPos * -1));
                            //construct positioning string
997         send6kCmd(cmd);

            //and write to 6k
998         finishStageMotion();
                                                                  //wait
            until stages finish moving
999
1000        posStatus = readCameraVal(&xPos, &yPos, &tPos);
                                    //get new positioning values
1001    }
1002
1003    tPos -= CI_tShift;
1004    sprintf(cmd, "4D%f: 4GO", tPos);
                                                  //adjust injector
        angle to most recent value
1005    send6kCmd(cmd);



1006
1007    return 0;
1008
1009
1010 /****** NOTE on "second chance" adjustments: sometimes the
    pattern recognition software has trouble with a recognizable
```

```
sample. I've found that a small positional
1011 shift suddenly renders the sample viable - not sure what this is
but it's internal to the In-Sight PatMax algorithm. Therefore
I've implemented a "second chance"
1012 in the event that a pattern is not recognized or stable, where the
software shifts the injector module a very small distance in the
x/y direction and then tries the
1013 alignment again. The number of "second chances" can be set by
initializing variable secondChance to whatever value is desired.
1014 ******/
1015 }
1016
1017
1018
1019 int FillSubstrate()
1020 {
1021     int injectLength, bfLength, TO_length, valvePct;
1022     double startTime, timeElapsed, valveVoltage;
1023
1024
1025
1026     GetCtrlVal (Hn_mainPanel, mainPanel_injectDuration, &
     injectLength);
1027     GetCtrlVal (Hn_mainPanel, mainPanel_backfillDuration, &bfLength
     );
1028     GetCtrlVal (Hn_mainPanel, mainPanel_topOffDuration, &TO_length);
1029     GetCtrlVal (Hn_mainPanel, mainPanel_numValvePct, &valvePct);
1030         valveVoltage = (double)valvePct * MAXPROPV / 100;
1031
1032
1033     setStatus("Opening flow valves...");
1034     setProgressBar (injectLength, 5);
                                                    //setup progress
     bar; update every 5 sec
1035
1036
1037     WriteToDigitalLine (1, "0", DIOports[2], 8, 0, 1);
                                  //pressurize DNA solutions
1038     WriteToDigitalLine (1, "0", DIOports[3], 8, 0, 1);
1039     rampVoltage(0, valveVoltage, 20);
1040     startTime = Timer ();
                                                                //note
     start time
1041     setStatus("Flowing solutions...");
1042     startProgressBar();
                                                                  //and
     start progress bar timer
```

```
1043
1044
1045        while((Timer() - startTime) < injectLength)
                                                //wait for user-specified
            duration
1046            ProcessSystemEvents();
1047
1048    //MessagePopup("Done", "Done with initial fill");
1049        WriteToDigitalLine (1, "0", DIOports[3], 8, 0, 0);
                                        //open DNA solns to vent
1050        Delay(10);
1051    //MessagePopup("Done", "Done with initial fill");
1052    // insert plugs
1053    /* send6kCmd("3MA1");
1054
            setStageMovement(2);
                        //set slow movement;
1055        send6kCmd("3V0.1");
1056        activeMotion =
            1;
            //anticipate upcoming motion
1057        send6kCmd("3D79.75:
            3GO");                                                  //and
            write to 6k
1058        finishStageMotion();
1059        send6kCmd("3MA1");
1060
1061        setStatus("Performing dead-end filling");
1062        setProgressBar (bfLength, 5);
1063        WriteToDigitalLine (1, "0", DIOports[3], 8, 0,
            1);                                 //reapply pressure to DNA solns
1064    // WriteToDigitalLine (1, "0", DIOports[1], 8, 0,
        1);                                 //apply back-pressure
1065        startTime = Timer
            ();
            //record start time
1066        startProgressBar();
1067
1068        while((Timer() - startTime) <
            bfLength)                                           //wait for
            user-specified duration
1069            ProcessSystemEvents();
1070    //MessagePopup("Done", "Done with initial fill");
1071
1072    /*   setStatus("Topping off chip"); 1073
        setProgressBar (TO_length, 5);
1074        WriteToDigitalLine (1, "0", DIOports[1], 8, 0,
```

```
        0);                                 //turn off back-pressure
1075    startTime = Timer ();
1076    startProgressBar();
1077
1078    while((Timer() - startTime) < TO_length)
1079        ProcessSystemEvents();
1080 */
1081
1082    WriteToDigitalLine (1, "0", DIOports[3], 8, 0, 0);
                                    //vent solutions again
1083    Delay(10);
1084    WriteToDigitalLine (1, "0", DIOports[2], 8, 0, 0);
                                    //switch low psi output to "vac"
1085
1086    ProcessSystemEvents();
1087    rampVoltage(0, 0, 10);
1088    WriteToDigitalLine (1, "0", DIOports[2], 8, 0, 0);
                                    //turn off pressure from both sides
1089    WriteToDigitalLine (1, "0", DIOports[3], 8, 0, 0);


1090
1091
1092    setStatus("");
1093
1094
1095    return 0;
1096 }
1097
1098
1099 int DisengageSubstrate()
1100 {
1101    double startTime;
1102    double VACTIME = 1;
1103    double extendVoltage = MAXPROPV;
1104
1105    //STOP POINTS: ADD TO CONFIG FILE!!!
1106    const double zStop1 = 65.0;
                                                        //hovering
1107    const double zStop2 = 76.1;
                                                        //start slow
       movement
1108    const double zStop25 = 78.8;
                                                       //injector
       pins in place
1109    const double zStop3 = 78.50; //79.5;
       //79.25;                                  //injector plugs in place
1110
```

```
1111        char cmd[80] = "";
1112
1113
1114        GetCtrlVal (Hn_mainPanel, mainPanel_pullout_Vac, &VACTIME);
1115
1116
1117        WriteToDigitalLine (1, "0", DIOports[3], 8, 0, 0);
                                        //make sure pressure/vent line is
            set to vent
1118    // WriteToDigitalLine (1, "0", DIOports[3], 8, 0,
        1);                             //turn on vacuum
1119    // startTime = Timer
        ();                                                              //note
        start time
1120    // while((Timer() - startTime) <
        VACTIME)                                        //wait for
        user-specified duration
1121    //       ProcessSystemEvents();
1122    // WriteToDigitalLine (1, "0", DIOports[3], 8, 0,
        0);                             //turn off vacuum
1123
1124    /*send6kCmd("3MA1"); 1125
        setStageMovement(1);
                    //set fast movement for pull-away
1126    activeMotion =
        1;
        //anticipate upcoming motion
1127    sprintf(cmd, "D 0, 0, %f: 3GO",
        -95);                                             //zero out x/y
        motion
1128    send6kCmd(cmd);
                    //and write to 6k
1129    finishStageMotion();
1130    MessagePopup("okay", "okay");
1131    setStageMovement(1);
                    //set fast movement for pull-away
1132    activeMotion =
        1;
        //anticipate upcoming motion
1133    sprintf(cmd, "D 0, 0, %f: 3GO",
        zStop2);                                          //zero out x/y
        motion
1134    send6kCmd(cmd);
                    //and write to 6k
1135    finishStageMotion();
1136    setStageMovement(2);
1137    activeMotion =
```

```
        1;
        //anticipate upcoming motion
1138    sprintf(cmd, "D 0, 0, %f: 3GO",
        zStop3);                                                //zero out x/y
        motion
1139    send6kCmd(cmd);
                    //and write to 6k
1140    finishStageMotion();
1141    */
1142
1143        setStatus("Ramping piston pressure");
1144        WriteToDigitalLine (1, "0", DIOports[1], 8, 0, 1);
                                        //open to psi
1145        WriteToDigitalLine (1, "0", DIOports[4], 8, 0, 1);
                                        //extend pistons
1146        rampVoltage(1, extendVoltage, 10);
                                                          //ramp pressure
1147


1148
1149
1150        setStatus("Disengaging substrate...");
                                                    //update front panel
1151        WriteToDigitalLine (1, "0", DIOports[2], 8, 0, 1);
                                        //open solutions to vent during
        pullout
1152        send6kCmd("3MA1");

            //set absolute positioning on Z axis
1153        setStageMovement(2);

            //start with slow disengagement to clear PDMS
1154        send6kCmd("3V0.1");
1155        activeMotion = 1;

            //anticipate upcoming motion
1156        sprintf(cmd, "D 0, 0, 76.8: 3GO");//%f: 3GO",
            zStop25-0.75);                      //zero out x/y motion
1157        send6kCmd(cmd);

            //and write to 6k
1158        finishStageMotion();
1159
1160
1161        WriteToDigitalLine (1, "0", DIOports[2], 8, 0, 0);
                                        //switch solutions to vacuum during
            final pull away
```

```
1162        setStageMovement(1);                                                    //set
            fast movement for pull-away
1163        activeMotion = 1;

            //anticipate upcoming motion
1164        sprintf(cmd, "D 0, 0, %f: 3GO", (zStop1-10));
                                                //zero out x/y motion
1165        send6kCmd(cmd);

            //and write to 6k
1166        finishStageMotion();
1167
1168 1169
1170        send6kCmd("3MA0");

            //return to relative pos on Z axis
1171        setStatus("Disengaging substrate... Done" );
1172
1173        WriteToDigitalLine (1, "0", DIOports[1], 8, 0, 0);
                                        //vent extension side
1174        Delay(2);
1175        WriteToDigitalLine (1, "0", DIOports[4], 8, 0, 0);
                                        //retract pistons
1176        WriteToDigitalLine (1, "0", DIOports[1], 8, 0, 1);
                                        //pressurize again
1177        Delay(3);

                //allow pressure to build to full
1178        setStatus("Retracting pistons");
1179        rampVoltage(1, 0, 10);
                                                                                //remove
            pressure
1180        WriteToDigitalLine (1, "0", DIOports[1], 8, 0, 0);
1181
1182
1183        setStatus("");
1184
1185        return 0;
1186    }
1187
1188
1189    int EngageSubstrate()
1190    {
1191        //STOP POINTS: ADD TO CONFIG FILE!!!
1192        const double zStop1 = 65.0;
```

```
                                                                    //hovering
1193        const double zStop2 = 76.1;
                                                                    //pressure
            plate engaged
1194        const double zStop25 = 78.8;
1195        const double zStop3 = 78.50; //79.5;
            //79.25;                                    //PDMS engaged
1196
1197        char cmd[80] = "";
1198
1199
1200        setStatus("Engaging substrate...");
                                                      //update front panel
1201        send6kCmd("3MA1");

            //set absolute positioning on Z axis
1202    ///send6kCmd("TPE");
1203
1204        setStageMovement(1);
                                                                        //set
            fast movement;
1205        activeMotion = 1;

            //anticipate upcoming motion
1206        sprintf(cmd, "D %f, %f, %f, 0: GO111x", CI_xShift, CI_yShift,
            zStop1);         //construct positioning string
1207        send6kCmd(cmd);

            //and write to 6k
1208        finishStageMotion();
                                                                        //wait
            until stages finish moving
1209
1210        setStageMovement(3);
                                                                        //set
            medium movement;
1211        activeMotion = 1;

            //anticipate upcoming motion
1212        sprintf(cmd, "D 0, 0, %f: 3GO", zStop2);
                                                  //zero out x/y motion
1213        send6kCmd(cmd);

            //and write to 6k
1214        finishStageMotion();
1215
1216        setStageMovement(2);
```

```
                                                                    //set
        slow movement;
1217    activeMotion = 1;

        //anticipate upcoming motion
1218    sprintf(cmd, "3D%f: 3GO", zStop25);
                                                    //construct
        positioning string
1219    send6kCmd(cmd);

        //and write to 6k
1220    finishStageMotion();
1221 /*MessagePopup("okay", "okay");
1222
        setStageMovement(2);
                    //set slow movement;
1223    activeMotion =
        1;
        //anticipate upcoming motion
1224    sprintf(cmd, "3D%f: 3GO",
        zStop3);                                          //construct
        positioning string
1225
        send6kCmd(cmd);
                    //and write to 6k
1226    finishStageMotion();
1227 */
1228
1229    send6kCmd("3MA0");

        //return to relative pos on Z axis
1230    setStatus("Engaging substrate... Done");
1231
1232    Delay(1);
1233
1234    return 0;
1235 }
1236
1237
1238 int processSlide(int slideNo)
1239 {
1240    const int CUTSCORE = 92;
1241
1242    int status = 2;
1243    int usrEvent, activePanel;
1244
1245 //status = 0;
```

```
1246
1247        updateStatus(slideNo, 0);
                                                                    //indicate
            active slide
1248            ProcessSystemEvents ();
                                                                      //
            update display
1249        status = PositionInjector(slideNo);
                                                                //position injector
            module
1250        while(status)

            //on error
1251        {    Hn_AlignPanel = LoadPanel (Hn_mainPanel, "alignScore.uir",
            scorePanel); // prompt for reduced alignment standards (cut
            score)
1252            activePanel = Hn_AlignPanel;
                                                                  // denote
            active panel
1253            InstallPopup (Hn_AlignPanel);
                                                                // install as
            popup
1254            while(activePanel == Hn_AlignPanel)
1255            {    ProcessSystemEvents ();
                                                                    // wait
            until it returns
1256                activePanel = GetActivePanel ();
1257            }
1258
1259            if(cutScore == CUTSCORE) return 1;
                                                                //   if cut score has
            not changed, stop processing slide
1260            status = PositionInjector(slideNo);
                                                                //   otherwise re-align
1261        }
1262        cutScore = CUTSCORE;

            //return cutScore to default
1263    //      if(status) return
        1;                                                          //on error,
        stop processing slide
1264        EngageSubstrate();
1265        FillSubstrate();
1266    //      status = check substrate
1267        DisengageSubstrate();
1268
1269
```

```
1270        Delay(1);
1271        updateStatus(slideNo, 2);
1272
1273        return 0;
1274    }
1275
1276    int checkAlignment(int slideNo)
1277    {
1278        const int CUTSCORE = 92;
1279
1280        int status = 2;
1281        int usrEvent, activePanel;
1282
1283    //status = 0;
1284
1285        updateStatus(slideNo, 0);
                                                                    //indicate
            active slide
1286              ProcessSystemEvents ();
                                                                      //
              update display
1287        status = PositionInjector(slideNo);
                                                          //position injector
            module
1288        if(cutScore == CUTSCORE){
1289            setStatus("Checks out");
1290            return 1;
1291        }
1292
1293        return 0;
1294    }
1295
1296
1297    int processPrimer()
1298    {
1299        int status = 2;
1300        int usrEvent;
1301        int injectLength, valvePct, rampTime;
1302        double startTime, timeElapsed, valveVoltage;
1303
1304    //status=0;
1305        slides[0][1] = 1;

            //ensure slide is "selected" if processed
1306        SetCtrlAttribute (Hn_mainPanel, slides[0][0], ATTR_FRAME_COLOR,
             activeSlideColor);
1307        updateStatus(0, 0);
```

```
        //activate primer slide
1308        ProcessSystemEvents ();
                                                                    //
            update display
1309    status = PositionInjector(0);
                                                                //position
        injector module
1310        if(status) return 1;
                                                                    //on
            error, stop processing slide
1311    EngageSubstrate();
1312
1313
1314
                //BEGIN MODIFIED "FILLSUBSTRATE()"
1315
1316    GetCtrlVal (Hn_mainPanel, mainPanel_primerDuration, &
        injectLength);            // read in user parameters
1317    GetCtrlVal (Hn_mainPanel, mainPanel_primerValvePct, &valvePct);
1318    GetCtrlVal (Hn_mainPanel, mainPanel_primerRampTime, &rampTime);
1319        valveVoltage = (double)valvePct * MAXPROPV / 100;
1320
1321
1322    setStatus("Priming solutions...");
1323    setProgressBar (injectLength, 5);
1324
1325    WriteToDigitalLine (1, "0", DIOports[2], 8, 0, 1);
                                  //open solenoid to pressurize vials
1326    WriteToDigitalLine (1, "0", DIOports[3], 8, 0, 1);
1327    rampVoltage(0, valveVoltage, rampTime);
                                              //slowly pressurize DNA
        solns
1328    startTime = Timer ();
                                                                    //note
        start time
1329    startProgressBar();
1330

1331    while((Timer() - startTime) < injectLength)
                                           //wait for user-specified
        duration
1332        ProcessSystemEvents();
1333
1334
1335    WriteToDigitalLine (1, "0", DIOports[3], 8, 0, 0);
                                  //open pressure system to vent
```

```
1336        Delay(10);
1337        ampVoltage(0, 0, 10);
                                                                    //close
            proportional valve
1338        WriteToDigitalLine (1, "0", DIOports[2], 8, 0, 0);
                                    //turn off solenoids to vials
1339 1340
1341        setStatus("");
1342
                    //END MODIFIED "FILLSUBSTRATE()"
1343 1344
1345        DisengageSubstrate();
1346
1347
1348        Delay(1);
1349        updateStatus(0, 2); 1350
            return 0;
1351    }
1352
1353 1354
1355    int CVICALLBACK cmdStartRun (int panel, int control, int event,
1356             void *callbackData, int eventData1, int eventData2)
1357    {
1358        int i, primerStatus, BlowoutStatus;
1359
1360
1361        if(event != EVENT_COMMIT) return 0;
1362
1363
1364        SetCtrlAttribute (Hn_mainPanel, mainPanel_cmdPauseRun,
            ATTR_DIMMED, 0);        //enable Pause & Abort buttons
1365        SetCtrlAttribute (Hn_mainPanel, mainPanel_cmdAbortRun,
            ATTR_DIMMED, 0);
1366
1367        i = 1;

                    //initialize to process second slide
1368        GetCtrlVal(Hn_mainPanel, mainPanel_chkUsePrimer, &primerStatus
            );              //get primer status
1369        if(primerStatus) processPrimer();
                                                        // if active, start
            priming
1370        else i--;
```

```
            //  otherwise begin processing at first slide
1371
1372        //Alex
1373        for(i = i; i < MAXSLIDES; i++)
1374            if(slides[i][1]) checkAlignment(i);
1375        //
1376        for(i = i; i < MAXSLIDES; i++)
1377            if(slides[i][1]) processSlide(i);
1378
1379
1380        GetCtrlVal(Hn_mainPanel, mainPanel_chkBlowout, &BlowoutStatus);
1381        if(BlowoutStatus) expungeSolutions(0,0,EVENT_COMMIT,NULL,0,0);
1382
1383
1384
1385        SetCtrlAttribute    (Hn_mainPanel, mainPanel_cmdNewRun,
            ATTR_DIMMED, 0);          //enable system reset button
1386        SetCtrlAttribute    (Hn_mainPanel, mainPanel_cmdStartRun,
            ATTR_DIMMED, 1);        //disable "start run" until reset occurs
1387        SetCtrlAttribute    (Hn_mainPanel, mainPanel_cmdPauseRun,
            ATTR_DIMMED, 1);        //disable Pause & Abort buttons
1388        SetCtrlAttribute    (Hn_mainPanel, mainPanel_cmdAbortRun,
            ATTR_DIMMED, 1);


1389
1390        return 0;
1391    }
1392
1393 1394
1395    int CVICALLBACK PauseRun (int panel, int control, int event,
1396            void *callbackData, int eventData1, int eventData2)
1397    {
1398        if(event != EVENT_COMMIT) return 0;
1399
1400        if(pauseFlag)

                            //if run is already paused
1401        {    pauseFlag = 0;

                        // release pause flag
1402            SetCtrlAttribute (Hn_mainPanel, mainPanel_cmdPauseRun,
                ATTR_LABEL_TEXT, "Pause Run");      // return button label
                to unpaused state
1403            SetCtrlAttribute (Hn_mainPanel, mainPanel_txtStatus,
                ATTR_CTRL_VAL, "");                    // update run status
1404        }
```

```
1405        else
                                        //else
1406        {    pauseFlag = 1;
                        // set pause flag
1407            SetCtrlAttribute (Hn_mainPanel, mainPanel_cmdPauseRun,
                ATTR_LABEL_TEXT, "Resume Run"); // change button label
1408            SetCtrlAttribute (Hn_mainPanel, mainPanel_txtStatus,
                ATTR_CTRL_VAL, "Run paused");          // update run
                status
1409        }
1410
1411        return 0;
1412    }
1413
1414 1415
1416    int CVICALLBACK AbortRun (int panel, int control, int event,
1417            void *callbackData, int eventData1, int eventData2)
1418    {
1419        if(event != EVENT_COMMIT) return 0;


1420
1421        if(!ConfirmPopup ("Abort Run", "Are you sure you want to abort
            this run?")) return 0;           //confirm cancel
1422
1423
1424        SetCtrlAttribute    (Hn_mainPanel, mainPanel_cmdPauseRun,
            ATTR_DIMMED, 1);        //disable Pause & Abort buttons
1425        SetCtrlAttribute    (Hn_mainPanel, mainPanel_cmdAbortRun,
            ATTR_DIMMED, 1);
1426        SetCtrlAttribute    (Hn_mainPanel, mainPanel_cmdNewRun,
            ATTR_DIMMED, 0);          //enable system reset button
1427        SetCtrlAttribute    (Hn_mainPanel, mainPanel_cmdStartRun,
            ATTR_DIMMED, 1);        //disable "start run" until reset occurs
1428
1429        return 0;
1430    }
1431
1432 1433
1434    int CVICALLBACK NewRun (int panel, int control, int event,
1435            void *callbackData, int eventData1, int eventData2)
1436    {
1437        int i, j;
```

```
1438
1439        if(event != EVENT_COMMIT) return 0;
1440
1441
1442
1443        for(i = 0; i < MAXSLIDES; i++)
                                                                    //for each slide
1444        {    if(slides[i][1])
                                                                                //
            that is present
1445             {    for(j = 2; j < 5; j++)
                                                                          //     check
                 each status icon
1446                      if(slides[i][j])

                         //     if present
1447                      {    DiscardCtrl (Hn_mainPanel, slides[i][j]);
                                                  //       delete icon ctrl
1448                          slides[i][j] = 0;

                             //       remove status flag
1449                      }
1450                 slides[i][1] = 0;

                                       //     deactivate slide
1451                 SetCtrlAttribute (Hn_mainPanel, slides[i][0],
                     ATTR_FRAME_COLOR, inactiveSlideColor);         //     remove
                     indicator
1452             }
1453        }
1454
1455 1456
1457        SetCtrlAttribute (Hn_mainPanel, mainPanel_cmdNewRun,
            ATTR_DIMMED, 1);          //disable system reset button
1458        SetCtrlAttribute (Hn_mainPanel, mainPanel_cmdStartRun,
            ATTR_DIMMED, 0);        //enable "start run"
1459
1460        return 0;
1461    }
1462
1463    int CVICALLBACK UpdateNFO (int panel, int control, int event,
1464             void *callbackData, int eventData1, int eventData2)
1465    {
1466        HRESULT err;
1467        float resultArr[3];
```

```
        //theta, y, x
1468    char results[80];
1469 //  char qString[80] = " ";
1470    int i = 0, j = 0;
1471    long qString = 0;
1472
1473    if(event != EVENT_COMMIT) return 0;
1474
1475    DS_Update   (DataSockets[0]);
1476    DS_Update   (DataSockets[1]);
1477    DS_Update   (DataSockets[2]);
1478    DS_Update   (DataSockets[3]);
1479    DS_Update   (DataSockets[4]);
1480
1481
1482    for(i = 0; i < 3; i++)
1483    {   while(qString != 192 && j++ < 5)
                                                  //192 = "Good"
1484            err = DS_GetAttrValue (DataSockets[i], "Quality",
                CAVT_LONG, &qString, sizeof(qString), NULL, NULL);
1485        if(j > 5) MessagePopup("Comm Error", "Could not retrieve
            data from OPC server!");
1486        else
1487        {   err = DS_GetDataValue (DataSockets[i], CAVT_FLOAT, &
            resultArr[i], sizeof(resultArr[0]), NULL, NULL);
1488            if(err < 0) CA_DisplayErrorInfo(DataSockets[i],
                "DataSocket Error", err, NULL);
1489        }
1490        j = 0; qString = 0;
1491    }
1492
1493
1494    sprintf(results, "x shift: %f\n y shift: %f\n t shift: %f\n" ,
        resultArr[2], resultArr[1], resultArr[0]);
1495    MessagePopup("Results", results);
1496
1497    return 0;
1498 }
1499
1500 1501
1502 int CVICALLBACK chooseJobFile (int panel, int control, int event,
1503          void *callbackData, int eventData1, int eventData2)
1504 {
1505     HRESULT err;
1506     int ringVal;
1507 // char fileNames[5][50] = {"", "40-pin.job", "100-pin.job"};
```

```
1508        char fileName[50];
1509    int i;
1510        if(event != EVENT_COMMIT) return 0;
1511
1512        GetCtrlVal (panel, control, &ringVal);
                                                    //generalized for use on
            other panels
1513        if(ringVal == -1)
1514        {    MessagePopup ("Using new job files", "Please consult the
            user's guide for instructions on how to add new job files" );
1515            return 0;
1516        }
1517        GetLabelFromIndex (panel, control, ringVal, fileName);
1518
1519
1520        err = DS_SetDataValue (DataSockets[5], CAVT_CSTRING, fileName,
            strlen(fileName), 0);            //set user-selected filename
1521        if(err < 0)

                //if error
1522        {    CA_DisplayErrorInfo(DataSockets[5], "DataSocket Error", err
            , NULL);       // display
1523            SetCtrlVal (Hn_mainPanel, mainPanel_CognexRing, 0);
                                        // reset ring
1524            return 0;
1525        }
1526
1527        SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, "Loading new
            job file..."); //update status
1528        ProcessSystemEvents();
1529
1530    i=   DS_Update(DataSockets[5]);
                                                              //else send
        update to Cognex server
1531        SetCtrlVal (Hn_mainPanel, mainPanel_txtStatus, "");


1532        ProcessSystemEvents();
1533
1534        return 0;
1535    }
1536
1537    int CVICALLBACK send6kCommand (int panel, int control, int event,
1538            void *callbackData, int eventData1, int eventData2)
1539    {
1540        char cmd[80];
1541        char errorMsg[120];
```

```
1542        char ts[2] = {(char)13, '\0'};
1543        int err = 0;
1544
1545 1546
1547        if(event != EVENT_COMMIT) return 0;
1548
1549
1550        GetCtrlVal (Hn_mainPanel, mainPanel_txtCmd, cmd);
1551        send6kCmd(cmd);
1552
1553        return 0;
1554    }
1555
1556    int CVICALLBACK EnableStage (int panel, int control, int event,
1557             void *callbackData, int eventData1, int eventData2)
1558    {
1559        int LEDval = 1;
1560        char sendString[20] = "DRIVExxx";
                                                          //default string to
          enable/disable drives
1561        char bitVals[2] = {'0','1'};
1562
1563        if(event != EVENT_COMMIT) return 0;
1564
1565        switch(control)

                        //determine which control called function
1566        {    case mainPanel_xEnable:

          //   for x stage
1567                 GetCtrlVal (Hn_mainPanel, mainPanel_xEnable, &LEDval);
                                                //     get status
1568                 sendString[5] = bitVals[LEDval];
                                                                        //
                 set status
1569                 SetCtrlAttribute (Hn_mainPanel, mainPanel_xIncrement,
                 ATTR_DIMMED, !LEDval);       //     set status on movement
                 icons
1570                 SetCtrlAttribute (Hn_mainPanel, mainPanel_xDecrement,
                 ATTR_DIMMED, !LEDval);       //     set status on movement
                 icons
1571                 break;
1572             case mainPanel_yEnable:

                 // for y stage
1573                 GetCtrlVal (Hn_mainPanel, mainPanel_yEnable, &LEDval);
```

```
                                                //     get status
1574                sendString[6] = bitVals[LEDval];
                                                                        //
                    set status
1575                SetCtrlAttribute (Hn_mainPanel,     mainPanel_yIncrement,
                    ATTR_DIMMED, !LEDval);      //     set status on movement
                    icons
1576                SetCtrlAttribute (Hn_mainPanel,     mainPanel_yDecrement,
                    ATTR_DIMMED, !LEDval);      //     set status on movement
                    icons
1577                break;


1578            case mainPanel_zEnable:

                    // for z stage
1579                GetCtrlVal (Hn_mainPanel, mainPanel_zEnable, &LEDval);
                                                //     get status
1580                sendString[7] = bitVals[LEDval];
                                                                        //
                    set status
1581                SetCtrlAttribute (Hn_mainPanel, mainPanel_zIncrement,
                    ATTR_DIMMED, !LEDval);      //     set status on movement
                    icons
1582                SetCtrlAttribute (Hn_mainPanel, mainPanel_zDecrement,
                    ATTR_DIMMED, !LEDval);      //     set status on movement
                    icons
1583                break;
1584        }
1585
1586        send6kCmd(sendString);
1587
1588        return 0;
1589    }
1590
1591    int CVICALLBACK manualMotion (int panel, int control, int event,
1592            void *callbackData, int eventData1, int eventData2)
1593    {
1594        float xyDistance = 0, zDistance = 0;
1595        char sendString[80];
1596
1597        if(event != EVENT_COMMIT) return 0;
1598
1599
1600        GetCtrlVal (Hn_mainPanel, mainPanel_xyDistance, &xyDistance);
                    //get distance values
1601        GetCtrlVal (Hn_mainPanel, mainPanel_zDistance, &zDistance);
```

```
1602
1603        switch(control)
1604        {

                    //determine which control called function
1605            case mainPanel_xIncrement:
                                                                // for positive
                x motion
                    sprintf(sendString, "1D-%3f: 1go", xyDistance);
                                        //     record distance & instigate
                    motion
1607                break;
1608                case mainPanel_xDecrement:
                                                                // for negative
                    x motion
1609                sprintf(sendString, "1D%3f: 1go", xyDistance);
                                        //     record distance, but add "-" sign
1610                break;
1611            case mainPanel_yIncrement:
1612                sprintf(sendString, "2D%3f: 2go", xyDistance);

1613                break;
1614            case mainPanel_yDecrement:
1615                sprintf(sendString, "2D-%3f:    2go", xyDistance);
1616                break;
1617            case mainPanel_zIncrement:
1618                sprintf(sendString, "3D-%3f:    3go", zDistance);
1619                break;
1620            case mainPanel_zDecrement:
1621                sprintf(sendString, "3D%3f:     3go", zDistance);
1622                break;
1623            case mainPanel_tIncrement:
1624                sprintf(sendString, "4D-%3f:    4go", zDistance);
1625                break;
1626            case mainPanel_tDecrement:
1627                sprintf(sendString, "4D%3f:     4go", zDistance);
1628                break;
1629        }
1630
1631        send6kCmd(sendString);
                                                                    //send
            string to 6k
1632
1633        return 0;
1634    }
1635
1636    int CVICALLBACK callHomeStages (int panel, int control, int event,
```

```
1637            void *callbackData, int eventData1, int eventData2)
1638    {
1639        if(event != EVENT_COMMIT) return 0;
1640
1641        HomeStages();
1642
1643        return 0;
1644    }
1645
1646 1647
1648
1649    int CVICALLBACK setDIO (int panel, int control, int event,

1650            void *callbackData, int eventData1, int eventData2)
1651    {

            //FIX THESE LABELS ON THE FRONT PANEL!!!
1652        int switchStatus;
1653
1654        if(event != EVENT_COMMIT) return 0;
1655
1656
1657        GetCtrlVal (Hn_mainPanel, control, &switchStatus);
                                  //get new value of calling control
1658        switch(control)

        //determine which control called function
1659        {    case mainPanel_bswBacklight:
1660                 WriteToDigitalLine (1, "0", DIOports[0],     8, 0,
                 switchStatus);
1661                 break;
1662             case mainPanel_bswLowpsi_vac:
1663                 WriteToDigitalLine (1, "0", DIOports[1],     8, 0,
                 switchStatus);
1664                 break;
1665             case mainPanel_bswLowpsi:
1666                 WriteToDigitalLine (1, "0", DIOports[2],     8, 0,
                 switchStatus);
1667                 break;
1668             case mainPanel_bswVacOn:
1669                 WriteToDigitalLine (1, "0", DIOports[3],     8, 0,
                 switchStatus);
1670                 break;
1671        }
1672
1673        return 0;
```

```
1674    }
1675
1676
1677
1678    int CVICALLBACK togglePropValve (int panel, int control, int event,
1679             void *callbackData, int eventData1, int eventData2)
1680    {
1681         int valvePct = 0;
1682         double valveVoltage = 0;
1683         int status;
1684
1685         if(event != EVENT_COMMIT) return 0;
1686
1687
1688         GetCtrlVal (Hn_mainPanel, control, &status);
1689         if(status)
1690         {    GetCtrlVal (Hn_mainPanel, mainPanel_numValvePct, &valvePct);
1691              valveVoltage = (double)valvePct * MAXPROPV / 100;
1692              rampVoltage(0, valveVoltage, 20);
1693         }

                      //ramp voltage up over 20 sec
1694         else
1695         {    rampVoltage(0, 0, 10);
1696         }

                      //ramp voltage down over 10 sec
1697
1698
1699         return 0;
1700    }
1701
1702 1703
1704    int CVICALLBACK quitProgram (int panel, int control, int event,
1705             void *callbackData, int eventData1, int eventData2)
1706    {    int i;
1707         if(event != EVENT_COMMIT) return 0;
1708
1709         writeConfig();
1710         HomeStages();
1711
1712         DS_DiscardObjHandle    (DataSockets[0]);
1713         DS_DiscardObjHandle    (DataSockets[1]);
1714         DS_DiscardObjHandle    (DataSockets[2]);
1715         DS_DiscardObjHandle    (DataSockets[3]);
1716         DS_DiscardObjHandle    (DataSockets[4]);
```

```
1717        DS_DiscardObjHandle (DataSockets[5]);
1718
1719        DisconnectFromTCPServer (sixK_TCP);
1720
1721        for(i = 0; i < 8; i++)
1722            WriteToDigitalLine (1, "0", i, 8, 1, 0);
1723        rampVoltage(0, 0.0, 5);
1724        rampVoltage(1, 0.0, 5);
1725
1726        QuitUserInterface(0);
1727
1728        return 0;
1729    }
1730
1731 1732
1733 1734
1735 1736
1737 1738
1739 1740
1741
1742    /*********************************************************************** 1743
       ***********************************************************************
1744
1745                          CALIBRATION     FUNCTIONS
1746
1747    ***********************************************************************
1748    ***********************************************************************/
1749
1750
1751
1752    int CVICALLBACK LaunchCalibration (int panel, int control, int
       event,
1753             void *callbackData, int eventData1, int eventData2)
1754    {
1755        if(event != EVENT_COMMIT) return 0;
1756
1757            send6kCmd("DRIVE1111");

                //activate all drives
1758            Hn_calibratePanel = LoadPanel (Hn_mainPanel,
                "calibration.uir", pnlCalib); //load panel
1759
```

```
1760            DisplayPanel (Hn_calibratePanel);
                                                                    //open panel
1761 1762
1763        return 0;
1764    }
1765
1766 1767
1768    int CVICALLBACK CmanualMotion (int panel, int control, int event,
1769            void *callbackData, int eventData1, int eventData2)
1770    {
1771        float xyDistance = 0, zDistance = 0;
1772        char sendString[80];
1773
1774        if(event != EVENT_COMMIT) return 0;
1775
1776
1777        GetCtrlVal (Hn_calibratePanel, pnlCalib_xyDistance, &xyDistance
            );        //get distance values
1778        GetCtrlVal (Hn_calibratePanel, pnlCalib_zDistance, &zDistance);
1779
1780
1781        switch(control)
1782        {

                //determine which control called function
1783            case pnlCalib_xIncrement:
                                                                    // for
                positive x motion
1784                sprintf(sendString, "1D-%3f: 1go", xyDistance);
                                        //    record distance & instigate
                motion
1785                break;
1786            case pnlCalib_xDecrement:
                                                                    // for
                negative x motion
1787                sprintf(sendString, "1D%3f: 1go", xyDistance);
                                        //    record distance, but add "-" sign
1788                break;
1789            case pnlCalib_yIncrement:
1790                sprintf(sendString, "2D%3f: 2go", xyDistance);

1791                break;
1792            case pnlCalib_yDecrement:
```

```
1793                sprintf(sendString, "2D-%3f:    2go", xyDistance);
1794                break;
1795            case pnlCalib_zIncrement:
1796                sprintf(sendString, "3D-%3f:    3go", zDistance);
1797                break;
1798            case pnlCalib_zDecrement:
1799                sprintf(sendString, "3D%3f:     3go", zDistance);
1800                break;
1801            case pnlCalib_tIncrement:
1802                sprintf(sendString, "4D-%3f:    4go", zDistance);
1803                break;
1804            case pnlCalib_tDecrement:
1805                sprintf(sendString, "4D%3f:     4go", zDistance);
1806                break;
1807        }
1808
1809        send6kCmd(sendString);                                          //send
        string to 6k
1810
1811        return 0;
1812    }
1813
1814
1815    int CVICALLBACK CreturnToMain (int panel, int control, int event,
1816            void *callbackData, int eventData1, int eventData2)
1817    {
1818        if(event != EVENT_COMMIT) return 0;
1819
1820        DiscardPanel (Hn_calibratePanel);
1821
1822        return 0;
1823    }
1824
1825
1826
1827
1828
1829    /************        ICD FUNCTIONS        *************/
1830
1831
1832    int CVICALLBACK CstartICD (int panel, int control, int event,
1833            void *callbackData, int eventData1, int eventData2)
1834    {
1835        int i = 0;
1836        int fileSize = 0;
1837        char entryName[80];
```

```
1838        FILE* jobFiles;
1839
1840
1841        if(event != EVENT_COMMIT) return 0;
1842
1843        Hn_ICDpanel = LoadPanel (Hn_calibratePanel, "calibration.uir",
            pnlICD);
1844        SetCtrlAttribute (Hn_ICDpanel, pnlICD_cmdGetCameraPos,
            ATTR_VISIBLE, 1);       //enable alignment start button
1845        SetCtrlAttribute (Hn_ICDpanel, pnlICD_cmdGetCameraPos_2,
            ATTR_VISIBLE, 0); //and disable record-&-continue button
1846
1847    /*
                    //zero out stage rotation
1848        activeMotion =
            1;
            //anticipate upcoming motion
1849        send6kCmd("4MA1: 4D0:
            4GO");                                                  //set
            absolute mode on axis 4 and go home
1850
            finishStageMotion();
                        //wait until stages finish moving
1851
            send6kCmd("4MA0");
                        //return to incremental mode on axis 4
1852    */
1853        send6kCmd("@DRIVE1");

            //enable all stages
1854        backlight(1);

            //and backlight
1855
1856        sprintf(entryName, "3MA1: 3d%f: 3go", imagingZ);
                                        //set absolute mode on axis 3 and
            go to imaging loc.
1857        send6kCmd(entryName);
                                                                        //move
            to imaging position
1858        finishStageMotion();
                                                                         //wait
            until stage finishes moving
1859        send6kCmd("3MA0");

            //return to incremental mode on axis 3
1860
```

```
1861
                    //populate jobfile list
1862        if(!(GetFileInfo (jobsFileName, &fileSize)))
                                            //if file doesn't exist
1863        {    sprintf(entryName, "There was a problem opening '%s'.
            Please \nensure that it is in this program's root directory" ,
            jobsFileName);
1864             MessagePopup ("Error", entryName);
1865             return 1;
1866        }
1867
1868        jobFiles = fopen (jobsFileName, "r");
1869        while(!feof (jobFiles))
                                                                  //while end
            of file not reached
1870        {    fgets (entryName, sizeof(entryName), jobFiles);
1871             if(entryName[strlen(entryName)-1] == '\n') entryName[strlen
                 (entryName)-1] = '\0';      // if CR present,strip it
1872             InsertListItem (Hn_ICDpanel, pnlICD_CognexRing, -1,
                 entryName, i++);                    // add it to calibration
                 panel Cognex ring
1873        }
1874
1875        GetCtrlVal (Hn_mainPanel, mainPanel_CognexRing, &i);
                               //get value from main panel
1876        SetCtrlVal (Hn_ICDpanel, pnlICD_CognexRing, i);
                                  //and set it on ICD panel
1877 1878
1879        DisplayPanel (Hn_ICDpanel);
1880
1881
1882        return 0;
1883    }
1884
1885    int CVICALLBACK CgetCameraPos (int panel, int control, int event,
1886             void *callbackData, int eventData1, int eventData2)
1887    {
1888
1889        const double xyTolerance = 0.025;
1890
1891        char cmd[80] = "";
1892        double xPos = .026, yPos = .026, tPos = 0;
1893        int posStatus = 0;
1894
1895
1896        if(event != EVENT_COMMIT) return 0;
```

```
1897
1898
1899        posStatus = readCameraVal(&xPos, &yPos, &tPos);
                                    //get offset values
1900        while((fabs(xPos) > xyTolerance) || (fabs(yPos) > xyTolerance))
                  //while out of spec
1901        {
1902            ProcessSystemEvents();
1903            if(posStatus)

                                            //if error generated, give
            message
1904            {    if(posStatus == 1) MessagePopup ("Alignment Error",
            "Unable to achieve stable alignment reading" );
1905                 if(posStatus == 2) MessagePopup ("Alignment Error",
                 "Unable to match pattern");
1906                 if(posStatus == 3) MessagePopup ("Alignment Error",
                 "Error communicating with vision OPC server" );
1907
1908                 MessagePopup("Action Required", "Unable to
                 automatically align pattern - please zero the
                 reference index values manually" );
1909                 goto CONTINUE;


1910             }
1911             setStatus("Performing fine adjustments to injector
                 position");
1912
1913             setStageMovement(2);
                                                                  //set slow
            motion for better precision
1914             activeMotion = 1;

                 //anticipate upcoming motion
1915             sprintf(cmd, "D %f, %f, 0, 0: @GO", xPos, (yPos * -1));
                           //construct positioning string
1916             send6kCmd(cmd);
                                                                        //and
                 write to 6k
1917             finishStageMotion();
                                                                  //wait
                 until stages finish moving
1918             ProcessSystemEvents();
1919
1920             posStatus = readCameraVal(&xPos, &yPos, &tPos);
                                    //get new positioning values
```

```
1921        }
1922
1923        MessagePopup("Alignment finalized", "Please press the button
            again to confirm that the alignment is correct" );
1924
1925        CONTINUE:
1926
1927        SetCtrlAttribute (Hn_ICDpanel, pnlICD_cmdGetCameraPos,
            ATTR_VISIBLE, 0);           //disable alignment start button
1928        SetCtrlAttribute (Hn_ICDpanel, pnlICD_cmdGetCameraPos_2,
            ATTR_VISIBLE, 1);         //and enable record-&-continue button
1929
1930
1931        return 0;
1932    }
1933
1934
1935    int CVICALLBACK CgetCameraPos2 (int panel, int control, int event,
1936             void *callbackData, int eventData1, int eventData2)
1937    {
1938        //STOP POINTS: ADD TO CONFIG FILE!!!
1939        const double zStop1 = 65.0;
                                                                  //hovering
1940        const double zStop2 = 76.1;
                                                                  //pressure plate
            engaged
1941        const double zStop3 = 79.90;
                                                                 //PDMS engaged
1942
1943
1944        char cmd[80] = ""; double tShift = 0;
1945        double xPos = 0.26, yPos = 0.26;
1946
1947        if(event != EVENT_COMMIT) return 0;
1948
1949
1950        while((xPos > 0.25) || (yPos > 0.25))
                                                 //if detected pattern is
            shifting
1951        {    GetCtrlVal(Hn_ICDpanel, pnlICD_currXpos, &xPos);
                                 // only grab theta value from properly
1952             GetCtrlVal(Hn_ICDpanel, pnlICD_currYpos, &yPos);
                                    // detected pattern
1953             GetCtrlVal(Hn_ICDpanel, pnlICD_currTpos, &tShift);
1954             Delay(0.5);
1955             ProcessSystemEvents();
1956        }
```

```
1957
1958        send6kCmd("TPE");

            //request encoder position; values read into globals
1959        Delay(1.0);

            //wait for response
1960        ProcessSystemEvents();
1961
1962        SetCtrlVal (Hn_ICDpanel, pnlICD_cameraX, encX);
                                    //copy values out of globals
1963        SetCtrlVal (Hn_ICDpanel, pnlICD_cameraY, encY);
1964        SetCtrlVal (Hn_ICDpanel, pnlICD_cameraT, tShift);
1965
1966
1967
1968        SetCtrlAttribute    (Hn_ICDpanel, pnlICD_cmdGetCameraPos,
            ATTR_DIMMED, 1);
1969        SetCtrlAttribute    (Hn_ICDpanel, pnlICD_cmdGetCameraPos_2,
            ATTR_DIMMED, 1);
1970        SetCtrlAttribute    (Hn_ICDpanel, pnlICD_cmdGetInjectorPos,
            ATTR_DIMMED, 0);
1971        SetCtrlAttribute    (Hn_ICDpanel, pnlICD_txtStep2, ATTR_DIMMED, 0);
1972        SetCtrlAttribute    (Hn_ICDpanel, pnlICD_txtStep1, ATTR_DIMMED, 1);
1973
1974        SetCtrlAttribute (Hn_ICDpanel, pnlICD_tmrPosUpdate,
            ATTR_ENABLED, 0);      //disable timer - no more updates needed
1975
1976 1977
1978        send6kCmd("3MA1");
                                                                    //set
            absolute positioning on Z axis
1979
1980        setStageMovement(1);
                                                                //set fast
            movement;
1981        activeMotion = 1;

            //anticipate upcoming motion
1982        sprintf(cmd, "D %f, %f, %f: GO111", CI_xShift, CI_yShift,
            zStop1);        //construct positioning string
1983        send6kCmd(cmd);
                                                                    //and
            write to 6k
1984        finishStageMotion();
                                                                //wait
```

```
        until stages finish moving
1985
1986        setStageMovement(3);
                                                                    //set
        medium movement;
1987        activeMotion = 1;

        //anticipate upcoming motion
1988        sprintf(cmd, "D 0, 0, %f: 3GO", zStop2);
                                            //zero out x/y motion
1989        send6kCmd(cmd);
                                                                        //and
        write to 6k
1990        finishStageMotion();
1991
1992        setStageMovement(2);
                                                                    //set slow
        movement;
1993
1994        send6kCmd("3MA0");
                                                                      //return
        to relative pos on Z axis
1995
1996        return 0;
1997    }
1998
1999
2000    int CVICALLBACK CgetInjectorPos (int panel, int control, int event,
2001             void *callbackData, int eventData1, int eventData2)
2002    {
2003        float injectorX = 0,
2004              injectorY = 0,
2005              injectorT = 0,
2006              cameraX     = 0, 2007
                  cameraY     = 0, 2008
                  cameraT     = 0;
2009
2010        if(event != EVENT_COMMIT) return 0;
2011
2012        GetCtrlVal (Hn_ICDpanel, pnlICD_cameraX, &cameraX);
                                //retrieve previously recorded values
2013        GetCtrlVal (Hn_ICDpanel, pnlICD_cameraY, &cameraY);
2014        GetCtrlVal (Hn_ICDpanel, pnlICD_cameraT, &cameraT);
2015
2016        send6kCmd("TPE");

        //request new encoder positions
```

```
2017        Delay(1.0);

            //wait for response
2018        ProcessSystemEvents();
2019
2020        injectorX = encX;
                                                                    //and
            copy into locals
2021        injectorY = encY;
2022        injectorT = encT;
2023
2024        CI_xShift = injectorX - cameraX;
                                                  //set global shift
            values
2025        CI_yShift = injectorY - cameraY;
2026        CI_tShift = cameraT - injectorT;
2027
2028        backlight(0);

            //turn off light
2029        MessagePopup("Calibration complete", "The camera-injector
            distance has been calibrated" );
2030
2031        DiscardPanel (Hn_ICDpanel);
2032
2033
2034        return 0;
2035    }
2036
2037 2038
2039 2040
2041    int CVICALLBACK CupdatePos (int panel, int control, int event,
2042             void *callbackData, int eventData1, int eventData2)
2043    {
2044        HRESULT err;
2045        float score, xPos, yPos, tPos;

2046        int i = 0, j = 0;
2047        long qString = 0;
2048
2049        if(event != EVENT_TIMER_TICK) return 0;
2050
2051
2052        DS_Update (DataSockets[0]);
2053        DS_Update (DataSockets[1]);
```

```
2054        DS_Update (DataSockets[2]);
2055        DS_Update (DataSockets[6]);
2056
2057
2058
2059
                    //first check if data is good
2060        while(qString != 192 && j++ < 5)
                                                          //192 = "Good"
2061            err = DS_GetAttrValue (DataSockets[6], "Quality", CAVT_LONG
                , &qString, sizeof(qString), NULL, NULL);
2062        if(j > 5) MessagePopup("Comm Error", "Could not retrieve data
            from OPC server!");
2063        else
2064        {   err = DS_GetDataValue (DataSockets[6], CAVT_FLOAT, &score,
            sizeof(score), NULL, NULL);
2065            if(err < 0) CA_DisplayErrorInfo(DataSockets[6],
                "DataSocket Error", err, NULL);
2066        }
2067
2068        if(score)

                                //if ptn found, read in x/y/theta shifts
2069        {   err = DS_GetDataValue (DataSockets[1], CAVT_FLOAT, &yPos,
            sizeof(score), NULL, NULL);       // and write to panel
2070            if(err < 0) CA_DisplayErrorInfo(DataSockets[1],
                "DataSocket Error", err, NULL);
2071
2072            err = DS_GetDataValue (DataSockets[2], CAVT_FLOAT, &xPos,
                sizeof(score), NULL, NULL);
2073            if(err < 0) CA_DisplayErrorInfo(DataSockets[2],
                "DataSocket Error", err, NULL);
2074
2075            err = DS_GetDataValue (DataSockets[0], CAVT_FLOAT, &tPos,
                sizeof(score), NULL, NULL);
2076            if(err < 0) CA_DisplayErrorInfo(DataSockets[0],
                "DataSocket Error", err, NULL);
2077
2078            SetCtrlAttribute (Hn_ICDpanel, pnlICD_currXpos, ATTR_DIMMED
                , 0);
2079            SetCtrlAttribute (Hn_ICDpanel, pnlICD_currYpos, ATTR_DIMMED
                , 0);
2080            SetCtrlAttribute (Hn_ICDpanel, pnlICD_currTpos, ATTR_DIMMED
                , 0);
2081
2082            if(fabs(xPos) <= 0.025) SetCtrlAttribute (Hn_ICDpanel,
                pnlICD_currXpos, ATTR_TEXT_COLOR, VAL_GREEN);
```

```
2083            if((fabs(xPos) > 0.025) && (fabs(xPos) < 0.1))
                SetCtrlAttribute (Hn_ICDpanel, pnlICD_currXpos,
                ATTR_TEXT_COLOR, 0x00C6C600);
2084            if(fabs(xPos) > 0.1) SetCtrlAttribute (Hn_ICDpanel,
                pnlICD_currXpos, ATTR_TEXT_COLOR, VAL_RED);
2085
2086            if(fabs(yPos) <= 0.025) SetCtrlAttribute (Hn_ICDpanel,
                pnlICD_currYpos, ATTR_TEXT_COLOR, VAL_GREEN);
2087            if((fabs(yPos) > 0.025) && (fabs(yPos) < 0.1))
                SetCtrlAttribute (Hn_ICDpanel, pnlICD_currYpos,
                ATTR_TEXT_COLOR, 0x00C6C600);
2088            if(fabs(yPos) > 0.1) SetCtrlAttribute (Hn_ICDpanel,
                pnlICD_currYpos, ATTR_TEXT_COLOR, VAL_RED);
2089
2090            if(fabs(tPos) <= 0.1) SetCtrlAttribute (Hn_ICDpanel,
                pnlICD_currTpos, ATTR_TEXT_COLOR, VAL_GREEN);
2091            if((fabs(tPos) > 0.1) && (fabs(tPos) < 1)) SetCtrlAttribute
                 (Hn_ICDpanel, pnlICD_currTpos, ATTR_TEXT_COLOR, 0x00C6C600
                );
2092            if(fabs(tPos) > 1) SetCtrlAttribute (Hn_ICDpanel,
                pnlICD_currTpos, ATTR_TEXT_COLOR, VAL_RED);
2093
2094            SetCtrlVal(Hn_ICDpanel,    pnlICD_currXpos, xPos);
2095            SetCtrlVal(Hn_ICDpanel,    pnlICD_currYpos, yPos);
2096            SetCtrlVal(Hn_ICDpanel,    pnlICD_currTpos, tPos);
2097            SetCtrlVal(Hn_ICDpanel,    pnlICD_ledPtnFound, 1);
2098        }
2099        else

                //else dim out everything.
2100        {   SetCtrlAttribute (Hn_ICDpanel, pnlICD_currXpos, ATTR_DIMMED
            , 1);
2101            SetCtrlAttribute (Hn_ICDpanel, pnlICD_currYpos, ATTR_DIMMED
                , 1);
2102            SetCtrlAttribute (Hn_ICDpanel, pnlICD_currTpos, ATTR_DIMMED
                , 1);
2103            SetCtrlVal(Hn_ICDpanel, pnlICD_ledPtnFound, 0);
2104        }
2105
2106 2107
2108        return 0;
2109    }
2110
2111    int CVICALLBACK Ccancel_ICD (int panel, int control, int event,
2112            void *callbackData, int eventData1, int eventData2)
2113    {
```

```
2114        if(event != EVENT_COMMIT) return 0;
2115
2116        backlight(0);

            //turn off light
2117        MessagePopup("Calibration cancelled", "The camera-injector
            distance was not calibrated");
2118
2119        DiscardPanel (Hn_ICDpanel);
2120        return 0;
2121    }
2122
2123 2124
2125 2126
2127    /*********     SLIDE LOCATION FUNCTIONS     **********/
2128
2129
2130    void initPosArray()
2131    {
2132        int i = 0, j = 0;
2133        int slideX, slideY, slideWidth, slideHeight;
2134        int labelX, labelY1, labelY2;
2135        char test[20];
2136        void* clickFunction = CslideClick;
2137
2138
2139        CslidePos[0][0] = pnlSL_slide0;
2140        CslidePos[1][0] = pnlSL_slide1;
2141        CslidePos[2][0] = pnlSL_slide2;
2142        CslidePos[3][0] = pnlSL_slide3;
2143        CslidePos[4][0] = pnlSL_slide4;
2144        CslidePos[5][0] = pnlSL_slide5;
2145        CslidePos[6][0] = pnlSL_slide6;
2146        CslidePos[7][0] = pnlSL_slide7;
2147        CslidePos[8][0] = pnlSL_slide8;
2148        CslidePos[9][0] = pnlSL_slide9;
2149        CslidePos[10][0] = pnlSL_slide10;
2150        CslidePos[11][0] = pnlSL_slide11;
2151        CslidePos[12][0] = pnlSL_slide12;
2152        CslidePos[13][0] = pnlSL_slide13;
2153        CslidePos[14][0] = pnlSL_slide14;
2154        CslidePos[15][0] = pnlSL_slide15;
2155        CslidePos[16][0] = pnlSL_slide16;
2156        CslidePos[17][0] = pnlSL_slide17;
2157
```

```
2158
2159        for(i = 0; i < MAXSLIDES; i++)
2160        {    GetCtrlAttribute (Hn_SLpanel, CslidePos[i][0],      ATTR_LEFT, &
            slideX);            //get slide position attributes
2161             GetCtrlAttribute (Hn_SLpanel, CslidePos[i][0],      ATTR_TOP, &
                 slideY);
2162             GetCtrlAttribute (Hn_SLpanel, CslidePos[i][0],      ATTR_WIDTH,
                 &slideWidth);
2163             GetCtrlAttribute (Hn_SLpanel, CslidePos[i][0],      ATTR_HEIGHT,
                  &slideHeight);
2164
2165             labelX = slideX + (int)((slideWidth - 40)/2);
                                                 //determine label placement
                 (label = 40px wide)
2166             labelY1 = slideY + (int)((slideHeight - 4)/3);
2167             labelY2 = slideY + 2* (int)((slideHeight - 4)/3);
2168
2169             CslidePos[i][1] = NewCtrl (Hn_SLpanel, CTRL_TEXT_MSG, "",
                 labelY1, labelX);        //create label controls
2170             CslidePos[i][2] = NewCtrl (Hn_SLpanel, CTRL_TEXT_MSG, "",
                 labelY2, labelX);
2171             for(j = 1; j < 3; j++)
2172             {    SetCtrlAttribute (Hn_SLpanel, CslidePos[i][j],
                 ATTR_SIZE_TO_TEXT, 0);                          //do not size
                 to text
2173                  SetCtrlAttribute (Hn_SLpanel, CslidePos[i][j],
                     ATTR_TEXT_JUSTIFY, VAL_CENTER_JUSTIFIED);        //center
                     text
2174                  SetCtrlAttribute (Hn_SLpanel, CslidePos[i][j],
                     ATTR_WIDTH, 40);                                 //set
                     width = 40
2175                  InstallCtrlCallback (Hn_SLpanel, CslidePos[i][j],
                     CslideClick, NULL);
2176                  sprintf(test, "%i.%i", i, j);
2177                  SetCtrlVal (Hn_SLpanel, CslidePos[i][j], test);
2178             }
2179        }
2180
2181        return;
2182    }
2183
2184 2185
2186    int CVICALLBACK ClaunchSL (int panel, int control, int event,
2187             void *callbackData, int eventData1, int eventData2)
2188    {
2189        int i = 0;
```

```
2190        char cXval[80], cYval[80];
2191
2192
2193        if(event != EVENT_COMMIT) return 0;
2194
2195
2196        Hn_SLpanel = LoadPanel (Hn_calibratePanel, "calibration.uir",
            pnlSL);
2197        SetCtrlVal (Hn_SLpanel, pnlSL_currSlide, pnlSL_slide1);
2198
2199        send6kCmd("@DRIVE1");
                                                                    //enable
            all stages
2200
2201        initPosArray();

            //initialize control IDs values
2202        for(i = 0; i < MAXSLIDES; i++)
                                                        //for each slide
2203        {    sprintf(cXval, "%3f", slidePos[i][0]);
                                        // copy current coordinates
2204             sprintf(cYval, "%3f", slidePos[i][1]);
2205             SetCtrlVal (Hn_SLpanel, CslidePos[i][1], cXval);
                                    // and update to screen
2206             SetCtrlVal (Hn_SLpanel, CslidePos[i][2], cYval);
2207        }
2208        SetCtrlVal(Hn_SLpanel, pnlSL_txtNewFilename, slidePosFile);
                    //update current file name
2209
2210        DisplayPanel (Hn_SLpanel);
2211
2212        return 0;
2213   }
2214
2215
2216   int CVICALLBACK CloadFile (int panel, int control, int event,
2217             void *callbackData, int eventData1, int eventData2)
2218   {
2219        int status = 0, i;
2220        char fileName[MAX_PATHNAME_LEN] = "";
2221        char inString[100] = "";
2222        float xVal, yVal;
2223        char cXval[20], cYval[20];
2224        FILE* SLfile;
2225        long fileSize;
2226
2227        if(event != EVENT_COMMIT) return 0;
```

```
2228
2229
2230        status = FileSelectPopup ("", "*.dat", "*.dat", "Select config
            file", VAL_LOAD_BUTTON, 0, 0, 1, 0, fileName);        //show file
            prompt
2231        if(status == 0) return 0;
                                                                  //if cancelled,
            exit
2232
2233        if(!(GetFileInfo (fileName, &fileSize)))
                                              //if file doesn't exist
2234        { sprintf(inString, "There was a problem opening '%s'!" ,
            fileName);
2235           MessagePopup ("Error", inString);
2236           return 1;
2237        }
2238
2239
2240        i = 0;
2241        SLfile = fopen (fileName, "r");
2242        while((!feof (SLfile)) && (i < MAXSLIDES))
                                           //while end of file not reached
2243        {    fgets (inString, sizeof(inString), SLfile);
                                     // get data
2244             sscanf (inString, "%f, %f\n", &xVal, &yVal);
                                        // extract info
2245             sprintf(cXval, "%3f", xVal); 2246
             sprintf(cYval, "%3f", yVal);
2247             SetCtrlVal (Hn_SLpanel, CslidePos[i][1], cXval);
                                    // and update to screen
2248             SetCtrlVal (Hn_SLpanel, CslidePos[i++][2], cYval);
2249        }
2250
2251        fflush (SLfile); 2252
             fclose (SLfile);

             //flush & close file
2253
2254        SplitPath (fileName, NULL, NULL, fileName);
                                           //strip out drive & path
2255        SetCtrlVal (Hn_SLpanel, pnlSL_txtNewFilename, fileName);
                           //update filename used
2256
2257        return 0;
2258    }
2259
2260
```

```
2261
2262  int CVICALLBACK CslideClick (int panel, int control, int event,
2263           void *callbackData, int eventData1, int eventData2)
2264  {
2265      int i = 0;
2266      int oldSlide = 0;
2267
2268      if(event != EVENT_LEFT_CLICK) return 0;
2269
2270
2271      GetCtrlVal (Hn_SLpanel, pnlSL_currSlide, &oldSlide);
                                                              //get prev
          slide's control ID
2272      SetCtrlAttribute (Hn_SLpanel, CslidePos[oldSlide][0],
          ATTR_FRAME_COLOR, inactiveSlideColor);             //remove highlight
2273      SetCtrlAttribute (Hn_SLpanel, CslidePos[oldSlide][1],
          ATTR_TEXT_BGCOLOR, inactiveSlideColor);
2274      SetCtrlAttribute (Hn_SLpanel, CslidePos[oldSlide][2],
          ATTR_TEXT_BGCOLOR, inactiveSlideColor);
2275
2276      while((control != CslidePos[i][0]) && (control != CslidePos[i][
          1]) && (control != CslidePos[i][2])) i++;        //find clicked
          slide
2277      SetCtrlVal (Hn_SLpanel, pnlSL_currSlide, i);
                                                                  //record
          new slide's array position
2278      SetCtrlAttribute (Hn_SLpanel, CslidePos[i][0], ATTR_FRAME_COLOR
          , activeSlideColor);
2279      SetCtrlAttribute (Hn_SLpanel, CslidePos[i][1],
          ATTR_TEXT_BGCOLOR, activeSlideColor);
2280      SetCtrlAttribute (Hn_SLpanel, CslidePos[i][2],
          ATTR_TEXT_BGCOLOR, activeSlideColor);
2281
2282      return 0;
2283  }
2284
2285
2286  int CVICALLBACK CsavePositions (int panel, int control, int event,
2287           void *callbackData, int eventData1, int eventData2)
2288  {
2289      int status = 0, i;
2290      char fileName[300] = "";
2291      char inString[100] = "";
2292      float xVal, yVal;
2293      char cXval[20], cYval[20];
2294      FILE* outFile;
2295      long fileSize;
```

```
2296
2297        if(event != EVENT_COMMIT) return 0;
2298
2299        i = 0;
2300        GetCtrlVal (Hn_SLpanel, pnlSL_txtNewFilename, fileName);
                                  //get file name
2301        strcpy(slidePosFile, fileName);
                                                          //copy to master
            file
2302        outFile = fopen (fileName, "w");
2303
2304        for(i = 0; i < MAXSLIDES; i++)
                                                           //for each slide
2305        {    GetCtrlVal (Hn_SLpanel, CslidePos[i][1], cXval);
                                      // get position
2306             GetCtrlVal (Hn_SLpanel, CslidePos[i][2], cYval);
2307
2308             slidePos[i][0] = atof(cXval);
                                                            //   and update
             in master array
2309             slidePos[i][1] = atof(cYval);
2310             fprintf(outFile, "%3f, %3f\n", slidePos[i][0], slidePos[i][
             1]);
2311        }
2312
2313        fflush (outFile); 2314
            fclose (outFile);

            //flush & close file
2315
2316        sprintf(slidePosFile, "%s", fileName);
                                                  //update file used.
2317        sprintf(inString, "File '%s' updated with new position values" ,
             fileName); //indicate to user
2318        MessagePopup("Calibration complete", inString);
2319
2320        DiscardPanel (Hn_SLpanel);
                                                                //unload panel
2321
2322        return 0;
2323    }
2324
2325    int CVICALLBACK CcancelSL (int panel, int control, int event,
2326             void *callbackData, int eventData1, int eventData2)
2327    {
2328        if(event != EVENT_COMMIT) return 0;
2329
```

```
2330        DiscardPanel (Hn_SLpanel);
                                                                    //no data
            overwritten, just bail out
2331
2332        return 0;
2333    }
2334
2335    int CVICALLBACK CsetSlidePos (int panel, int control, int event,
2336             void *callbackData, int eventData1, int eventData2)
2337    {
2338        int slideNo = 0;
2339        char cXval[20], cYval[20];
2340
2341        if(event != EVENT_COMMIT) return 0;
2342
2343
2344        GetCtrlVal (Hn_SLpanel, pnlSL_currSlide, &slideNo);
                                    //get slide number
2345
2346        send6kCmd("TPE");

            //request encoder positions
2347        Delay(1.0);

                //wait for response
2348        ProcessSystemEvents();
2349
2350        sprintf(cXval, "%3f", encX);
                                                                //copy into
            locals
2351        sprintf(cYval, "%3f", encY);
2352        SetCtrlVal (Hn_SLpanel, CslidePos[slideNo][1], cXval);
                                  // and update to screen
2353        SetCtrlVal (Hn_SLpanel, CslidePos[slideNo][2], cYval);
2354
2355
2356        return 0;
2357    }
2358
2359
2360    int CVICALLBACK CposCalc (int panel, int control, int event,
2361             void *callbackData, int eventData1, int eventData2)
2362    {
2363        int i, j;
2364        int slideNo = 0;
2365        float xShift, yShift;
2366        float xShiftTotal, yShiftTotal;
```

```
2367        float baseX, baseY;
2368        char cXshift[20], cYshift[20];
2369        int ROWS = 3, COLS = 6;
2370
2371
2372        if(event != EVENT_COMMIT) return 0;
2373
2374
2375        GetCtrlVal (Hn_SLpanel, pnlSL_currSlide, &slideNo);
                                        //get slide number
2376        GetCtrlVal (Hn_SLpanel, pnlSL_xShift, &xShift);
                                            //get shift values
2377        GetCtrlVal (Hn_SLpanel, pnlSL_yShift, &yShift);

2378        GetCtrlVal (Hn_SLpanel, CslidePos[slideNo][1], cXshift);
                                //recover x,y pos
2379        GetCtrlVal (Hn_SLpanel, CslidePos[slideNo][2], cYshift);
2380        baseX = atof(cXshift);
2381        baseY = atof(cYshift);
2382
2383
2384        if((xShift == 0) || (yShift == 0))
2385        {    MessagePopup("Data Error", "Error - x & y shift values
            must be non-zero!");
2386             return 0;
2387        }
2388
2389        for(i = 0; i < MAXSLIDES; i++)
                                                                //for each slide
2390        {    xShiftTotal = baseX - (((int)(i/COLS)-(int)(slideNo/COLS))*
            xShift);       // calculate differential based on relative
            position
2391             yShiftTotal = baseY + (((int)(i%COLS)-(int)(slideNo%COLS))*
                 yShift);
2392             sprintf(cXshift, "%f", xShiftTotal);
                                                    // convert to strings
2393             sprintf(cYshift, "%f", yShiftTotal);
2394             SetCtrlVal (Hn_SLpanel, CslidePos[i][1], cXshift);
                                     // store values
2395             SetCtrlVal (Hn_SLpanel, CslidePos[i][2], cYshift);
2396        }
2397
2398
2399        return 0;
2400   }
2401
2402
```

```
2403
2404
2405
2406  /*********       FOCAL LENGTH FUNCTIONS        **********/
2407
2408
2409
2410
2411  int CVICALLBACK ClaunchFL (int panel, int control, int event,
2412           void *callbackData, int eventData1, int eventData2)
2413  {
2414      int i = 0;
2415
2416      if(event != EVENT_COMMIT) return 0;
2417
2418
2419      Hn_FLpanel = LoadPanel (Hn_calibratePanel, "calibration.uir",
          pnlFL);
2420
2421      backlight(1);
2422
2423      DisplayPanel (Hn_FLpanel);
2424      return 0;
2425  }
2426
2427
2428  int CVICALLBACK CsefZdistance (int panel, int control, int event,
2429           void *callbackData, int eventData1, int eventData2)
2430  {
2431      if(event != EVENT_COMMIT) return 0;
2432
2433      send6kCmd("TPE");

          //request encoder positions
2434      Delay(1.0);

          //wait for response
2435      ProcessSystemEvents();
2436
2437      imagingZ = encZ;

          //record z value
2438
2439      backlight(0);

          //turn off backlight
2440      DiscardPanel (Hn_FLpanel);
```

```
                                                                    //and unload
        module
2441
2442        return 0;
2443    }
2444
2445    int CVICALLBACK CcancelFL (int panel, int control, int event,
2446            void *callbackData, int eventData1, int eventData2)
2447    {
2448        if(event != EVENT_COMMIT) return 0;
2449
2450        backlight(0);
2451        DiscardPanel (Hn_FLpanel);
2452
2453        return 0;
2454    }
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464    int CVICALLBACK goChip (int panel, int control, int event,
2465            void *callbackData, int eventData1, int eventData2)
2466    {
2467        float x, y;
2468        char goString[80];
2469
2470        if(event != EVENT_COMMIT) return 0;
2471
2472        GetCtrlVal (Hn_mainPanel, mainPanel_stdXshift, &x);
2473        GetCtrlVal (Hn_mainPanel, mainPanel_stdYshift, &y);
2474
2475        sprintf(goString, "D -%f, -%f, 140", x, y);
2476        send6kCmd(goString);
2477        send6kCmd("GO111");
2478
2479        return 0;
2480    }
2481
2482    int CVICALLBACK goFocus (int panel, int control, int event,
2483            void *callbackData, int eventData1, int eventData2)
2484    {
2485        float x, y;
```

```
2486        char goString[80];
2487
2488        if(event != EVENT_COMMIT) return 0;
2489
2490        GetCtrlVal (Hn_mainPanel, mainPanel_stdXshift, &x); 2491
            GetCtrlVal (Hn_mainPanel, mainPanel_stdYshift, &y);
2492
2493        sprintf(goString, "D %f, %f, -150", x, y);
2494        send6kCmd(goString);
2495        send6kCmd("GO111");
2496
2497        return 0;
2498    }
2499
2500
2501    int CVICALLBACK eStop (int panel, int control, int event,
2502            void *callbackData, int eventData1, int eventData2)
2503    {
2504        if(event != EVENT_COMMIT) return 0;
2505
2506        send6kCmd("!@S");
2507        activeMotion = 0;
2508
2509        return 0;
2510    }
2511
2512
2513    int CVICALLBACK DisengageInjector (int panel, int control, int
    event,
2514            void *callbackData, int eventData1, int eventData2)
2515    {
2516        if(event != EVENT_COMMIT) return 0;
2517            DisengageSubstrate();
2518        return 0;
2519    }
2520
2521    int CVICALLBACK cmdSpearIt (int panel, int control, int event,
2522            void *callbackData, int eventData1, int eventData2)
2523    {
2524        int i = 0;
2525        int status = 2;
2526        int usrEvent, slideNo;
2527
2528        if(event != EVENT_COMMIT) return 0;
2529
2530
2531        while((slides[i][1] == 0) && (i++ < MAXSLIDES));
```

```
2532        slideNo = i;
2533
2534        updateStatus(slideNo, 0);
                                                                    //indicate
            active slide
2535            ProcessSystemEvents ();
                                                                      //   update
            display
2536        status = PositionInjector(slideNo);
                                                        //position fluid module
2537            if(status) return 1;
                                                                          //on
            error, stop processing slide
2538        EngageSubstrate();
2539
2540
2541
2542        Delay(1);
2543        updateStatus(slideNo, 2);
2544
2545

2546        slides[slideNo][1] = 0;
                                                                    //deselect chip
2547        SetCtrlAttribute (Hn_mainPanel, slides[slideNo][0],
            ATTR_FRAME_COLOR, inactiveSlideColor);

2548        for(i = 1; i < 5; i++)
2549        {    if(slides[slideNo][i])
2550                 DiscardCtrl (Hn_mainPanel, slides[slideNo][i]);
                                        //       delete icon ctrl
2551             slides[slideNo][i] = 0;
2552        }
2553
2554
2555        return 0;
2556    }
2557
2558    int CVICALLBACK togglePistonValve (int panel, int control, int
            event,
2559             void *callbackData, int eventData1, int eventData2)
2560    {
2561        int valvePct = 0;
2562        double valveVoltage = 0;
2563        int status;
2564
2565        if(event != EVENT_COMMIT) return 0;
```

```
2566
2567        GetCtrlVal (Hn_mainPanel, control, &status);
                                        //determine if switch on or off
2568        if(status)

            //if on
2569        {    GetCtrlVal (Hn_mainPanel, mainPanel_numPistonPct, &valvePct
            );        // get valve percentage
2570             valveVoltage = (double)valvePct * MAXPROPV / 100;
                                  // convert to scaled voltage
2571             rampVoltage(1, valveVoltage, 15);
                                                    // update channel;
                 ramp over 15 sec
2572        }
2573        else

                 //if off
2574        {    rampVoltage(1, 0, 10);
                                                                //   update
            channel with 0 volts over 10 sec
2575        }
2576
2577
2578        return 0;
2579    }
2580
2581    int CVICALLBACK switchPiston (int panel, int control, int event,
2582             void *callbackData, int eventData1, int eventData2)
2583    {
2584        int status = 0;
2585
2586
2587        if(event != EVENT_COMMIT) return 0;
2588
2589
2590        GetCtrlVal(Hn_mainPanel, control, &status);
2591        WriteToDigitalLine (1, "0", DIOports[1], 8, 0, 0);
                                  //vent current direction
2592        Delay(1);
2593        WriteToDigitalLine (1, "0", DIOports[4], 8, 0, status);
                             //switch to opposite direction
2594        WriteToDigitalLine (1, "0", DIOports[1], 8, 0, 1);
                                  //pressurize
2595        setStatus("Allowing pistons to settle..." );
2596        Delay(5);
```

```
2597        setStatus("");
2598
2599
2600        return 0;
2601    }
2602
2603    int CVICALLBACK togglePrimer (int panel, int control, int event,
2604             void *callbackData, int eventData1, int eventData2)
2605    {
2606        int status = 0;
2607        int newColor;
2608
2609
2610        if(event != EVENT_COMMIT) return 0;
2611
2612
2613        GetCtrlVal(Hn_mainPanel, control, &status);
                                          //get check status
2614
2615        SetCtrlAttribute(Hn_mainPanel,     mainPanel_primerOutline,
            ATTR_VISIBLE, status);
2616        SetCtrlAttribute(Hn_mainPanel,     mainPanel_primerDuration,
            ATTR_DIMMED, !status);
2617        SetCtrlAttribute(Hn_mainPanel,     mainPanel_primerValvePct,
            ATTR_DIMMED, !status);
2618        SetCtrlAttribute(Hn_mainPanel,     mainPanel_primerRampTime,
            ATTR_DIMMED, !status);
2619
2620
2621        slides[0][1] = (status)?1:0;
                                                          // deactivate slide
2622        newColor = (status)?activeSlideColor : inactiveSlideColor;
2623             SetCtrlAttribute (Hn_mainPanel, slides[0][0],
                 ATTR_FRAME_COLOR, newColor);
2624 2625
2626        return 0;
2627    }
2628
2629 2630
2631 2632
2633 2634
2635    /*********      ALIGNMENT OVERRIDE FUNCTIONS       **********/
2636
```

```
2637
2638
2639
2640  int CVICALLBACK editCmdTime (int panel, int control, int event,
2641           void *callbackData, int eventData1, int eventData2)
2642  {
2643      int newTime;
2644      char txtLabel[20];
2645
2646
2647      GetCtrlVal (Hn_AlignPanel, scorePanel_numTimeRemaining, &
         newTime);        //get latest time
2648      if(newTime <= 0) FakeKeystroke (VAL_MENUKEY_MODIFIER | 'S');
                  //if time expired, default to cancel button
2649
2650      sprintf(txtLabel, "__Skip chip (%i)", newTime);
                                  //else print new button label
2651      SetCtrlAttribute (Hn_AlignPanel, scorePanel_cmdSkipChip,
         ATTR_LABEL_TEXT, txtLabel);
2652      SetCtrlVal (Hn_AlignPanel, scorePanel_numTimeRemaining, --
         newTime);       //record new time
2653
2654
2655      return 0;
2656  }
2657
2658  int CVICALLBACK cmdCancelAlign (int panel, int control, int event,
2659           void *callbackData, int eventData1, int eventData2)
2660  {
2661      if(event != EVENT_COMMIT) return 0;
2662
2663
2664      RemovePopup (1);
2665      DiscardPanel (Hn_AlignPanel);
2666
2667      return 0;
2668  }
2669
2670  int CVICALLBACK cmdRealign (int panel, int control, int event,
2671           void *callbackData, int eventData1, int eventData2)
2672  {
2673      int newCutScore;
2674
2675      if(event != EVENT_COMMIT) return 0;
2676
2677      GetCtrlVal (Hn_AlignPanel, scorePanel_sldNewScore, &newCutScore
         );        //get user value
```

```
2678        cutScore = newCutScore;                                    //and assign
            to global
2679
2680        RemovePopup (1);
2681        DiscardPanel (Hn_AlignPanel);
2682        return 0;
2683    }
2684
2685 2686
2687 2688
2689 2690
2691 2692
2693 2694
2695 2696
2697 2698
2699 2700
2701 2702
2703 2704
2705 2706
2707 2708
2709 2710
2711 2712
2713
2714    ////////////////////////////////////////////////////////////////
2715    ////////////////////// LEGACY CODE /////////////////////////////
2716    ////////////////////////////////////////////////////////////////
2717
2718
2719
2720
2721    int PosInjector2(int slideNo, double* myTpos)
2722    {
```

```
2723        const double xyTolerance = 0.025;
2724
2725        char cmd[80] = "";
2726        double xPos = .026, yPos = .026, tPos = 0;
2727        int posStatus = 0;
2728
2729
2730
2731        setStageMovement(1);
                                                                    //move to
            slide imaging position
2732        send6kCmd("MA1111");
                                                                    //set
            absolute positioning mode on all axes
2733            activeMotion = 1;

                //anticipate upcoming motion
2734            sprintf(cmd, "D %f, %f, %f, 0: @GO", slidePos[slideNo][0],
                slidePos[slideNo][1], imagingZ);          //construct
                positioning string
2735            send6kCmd(cmd);
                                                                        //and
                write to 6k
2736            finishStageMotion();
                                                                    //wait
                until stages finish moving
2737        send6kCmd("@MA0");

            //restore relative positioning
2738
2739
2740        posStatus = readCameraVal(&xPos, &yPos, &tPos);
                                    //get offset values
2741        while((fabs(xPos) > xyTolerance) || (fabs(yPos) > xyTolerance))
                  //while out of spec
2742        {
2743            if(posStatus)

                //if error generated, give message
2744            {    if(posStatus == 1) MessagePopup ("Alignment Error",
                "Unable to achieve stable alignment reading" );
2745                 if(posStatus == 2) MessagePopup ("Alignment Error",
                    "Unable to match pattern");
2746                 if(posStatus == 3) MessagePopup ("Alignment Error",
                    "Error communicating with vision OPC server" );
2747
2748                 updateStatus(slideNo, 4);
```

```
                                                                    //add error icon
2749    //            updateStatus(slideNo,
        0);                                                 //recover "active" color
2750                  return 1;


2751              }
2752              setStatus("Performing fine adjustments to injector
                  position");
2753
2754          setStageMovement(2);
                                                                          //set slow
          motion for better precision
2755              activeMotion = 1;

                  //anticipate upcoming motion
2756              sprintf(cmd, "D %f, %f, 0, 0: @GO", xPos, (yPos * -1));
                                  //construct positioning string
2757              send6kCmd(cmd);
                                                                                //and
                  write to 6k
2758              finishStageMotion();
                                                                          //wait
                  until stages finish moving
2759
2760              posStatus = readCameraVal(&xPos, &yPos, &tPos);
                                        //get new positioning values
2761          }
2762
2763
2764
2765
2766          tPos -= CI_tShift;
2767          *myTpos = tPos;
2768    // sprintf(cmd, "4D%f: 4GO",
        tPos);                                              //adjust injector
        angle to most recent value
2769    //
        send6kCmd(cmd);


2770
2771          return 0;
2772    }
2773
2774
2775    int PosInject(int slideNo)
2776    {
```

```
2777        int MAXTRIES = 5;
2778        int posStatus, i;
2779        char cmd[80] = "";
2780        double xPos2, yPos2, tPos2;
2781        double yShift = 18;
2782        double arcTan;
2783        double tPos = 0, newTpos = 0;
2784
2785
2786        if(PosInjector2(slideNo, &newTpos)) return 1;
2787
2788        sprintf(cmd, "2V5: 2D18: 2GO");
2789        send6kCmd(cmd);
2790        posStatus = readCameraVal(&xPos2, &yPos2, &tPos2);
2791        i = 0;
2792
2793        while((i++ < MAXTRIES) && (posStatus))
                                                  //allow five tries to read
            value
2794        {    sprintf(cmd,"2D0.25: 2GO");
                                                        // perform small
            shift each time
2795             send6kCmd(cmd);
2796             yShift += 0.25;
2797             posStatus = readCameraVal(&xPos2, &yPos2, &tPos2);
2798        }
2799        if((i >= MAXTRIES) && (posStatus)) return 1;
2800
2801        arcTan = (xPos2) / (yShift + yPos2);
2802        tPos = atan(arcTan);
2803        tPos *= 180 / 3.141592;
2804    tPos += 0.21; //temp to account for deviation
2805
2806        PosInjector2(slideNo, &newTpos);
2807    //  tPos -= CI_tShift;
2808        sprintf(cmd, "4D%f: 4GO", newTpos);
                                                    //adjust injector angle
            to most recent value
2809        send6kCmd(cmd);
2810
2811        return 0;
2812    }
2813
2814
2815
2816
```

Chapter 5

An Integrated Hardware and Software System for Automating Microfluidics

5.1 Introduction

Since its introduction almost three decades ago, the field of microfluidics has experienced exponential growth and development[1,2]. As wide-scale adoption continues, there is a need to advance the infrastructure surrounding these devices, i.e. the hardware that controls and powers such chips. At the commercial level, dozens of companies have launched mature, fully integrated and automated products based around microfluidics platforms. In contrast, microfluidic work in academic labs remains a largely manual affair, due in part to its developmental nature, and due in part to its more transient ultimate goals. However, as years of individual component engineering have given way to complex, integrated chips focused on obtaining scientific results, there is a strong need to facilitate and automate their operation.

Existing efforts towards automation are sharply demarcated between the two major approaches to microfluidics: classical “continuous flow” and digital microfluidics. The latter technique shuttles exposed droplets of fluid across patterned electrodes on a planar surface

using electrowetting on dielectric (EWOD)[3] or dielectrophoresis (DEP)[4] techniques. Because the fluids are controlled with electronic impulses, this branch of microfluidics has been subject to significant automation with regards to both chip design and chip control [5-7]. Conversely, the infrastructure surrounding continuous flow systems, wherein fluids flow through enclosed microchannels of rigid architecture, remains poorly developed despite a much longer history and significantly higher adoption. Here, external pressure sources are generally required at each of the fluidic chip's inputs to drive reagent flow or to actuate integrated valves[8-10] that dynamically guide those reagents on chip. Applications that require tight control over flow rates employ syringe pumps which confer some limited programmability, but more often a gaseous pressure source is supplied to each input via manually operated toggle switches. As chip complexity increases, so too does the number of inputs, and manual operation becomes progressively more cumbersome, error-prone, and generally untenable. In many such cases, manual toggles are replaced with electronically activated solenoid valves which are then coordinated via custom Labview or Matlab scripts[11-14]. However, such automation routines are cumbersome to code, are specific to the chip at hand, and lack general applicability. Likewise, hardware implementations are exclusively "home brew" and lack standardization, as commercial solutions are rare.

The few concerted efforts towards automating continuous flow microfluidics center upon software abstraction[15-18]. Here, the aim is to allow users to issue a string of basic fluidic tasks via a computer program (e.g. "mix reagent 2 + reagent 5", "discard reagent 4", etc.) without having explicit knowledge of the underlying fluidic architecture. A compiler processes the commands and autonomously coordinates the appropriate valve actuations to accomplish the

desired operations; by stringing together multiple commands, a user can easily generate an automation routine. While these efforts to introduce abstraction to microfluidics are borne of sound principle, in practice they are excessive at the academic level, and they have thus failed to gain traction with research groups. Among the strengths of microfluidics is that new designs are easy to prepare and implement, and a dedicated circuit is inevitably more efficient than a generalized one. A more practical approach is to allow the user to fabricate a custom circuit, but then facilitate its control via software. Unfortunately, efforts in this direction[19,20] have thus far met with little fanfare.

Herein, we introduce an integrated software and hardware package aimed at facilitating and automating laboratory-scale microfluidics experiments. The software component features an intuitive graphical user interface (GUI) that affords the user facile control of single or multiple valves in the visual context of their microfluidic circuit. Predetermined configurations for multiple valves may also be recorded and arranged to create automation routines. The software is tailored to work with a custom, USB-driven hardware box which houses and controls up to 64 solenoid valves for chip control. We further adapt this hardware to create a high-throughput system capable of running automation routines on multiple fluidic chips simultaneously and asynchronously. Finally, we detail the transformation of our previously developed blood chip into a “one-touch” analysis system suitable for clinical trials through the use of these automation technologies.

5.2 Methods & Materials

5.2.1 Software.

Software for chip control and automation was developed in the National Instruments' Labwindows CVI programming environment. Hardware communication is handled by a pair of functions within the main program for the sake of simplicity and convenience; a wider release of this software would isolate these to create a formal device driver. A separate program was coded for the "one touch" system which features a drastically simplified user interface that obscures detailed valve configurations from non-technical users. This latter program also contains code that allows multiple instances of a chip to run simultaneously and asynchronously. Source code for both programs is provided in the appendix.

5.2.2 Hardware

The solenoid control hardware consists of a PCB "motherboard" mounted inside a simple metal chassis. Inputs are limited to a 24V DC power supply, a USB connection, and several barbed tubing ports to supply pressurized air and/or vacuum as needed, keeping the entire ensemble compact and portable. A single panel on the chassis incorporates 64 embedded stainless steel pins (23-gage, New England Small Tube), each of which corresponds to a solenoid within. The user simply connects these pins directly to their microfluidic chip via standard Tygon tubing.

Within the chassis, solenoids (LHDA2421111H, Lee Company) are screw-mounted onto custom-fabricated manifolds in groups of eight. Each manifold features a single #10-32 threaded

pressure input which is distributed to the normally-closed (NC) ports of mounted solenoids. The manifold also incorporates a set of 23 gage pins that correspond to the solenoids' common (C) ports, and these ultimately connect to the user's microfluidic chip via the chassis panel. Assembled manifolds are installed by simply slotting them into corresponding sockets on the PCB motherboard via the solenoids' electrical contact pins, which protrude from beneath the assembly. Figure 5.5.1a depicts the solenoid control hardware populated with several manifolds.

The PCB motherboard is built around a USB chip (DLP-232PC, DLP Technologies) which provides fourteen digital I/O (DIO) lines that are multiplexed to control up to 64 solenoids, as shown schematically in Figure 5.5.2b. Briefly, eight lines are used to form a common data bus which can configure all the solenoids for a single manifold simultaneously. To ensure that only one manifold is modified at a time, a set of octal D-latches is placed between the bus and the solenoids; these chips record the bus state when enabled, ignore it when disabled, and otherwise continuously output their last recorded state to the solenoids. Thus, in order to reconfigure a solenoid, the hardware driver simply loads the data bus appropriately and then briefly enables the relevant octet of D-latches. Four of the USB chip's DIO lines are dedicated to a 3-bit multiplexer that accomplishes the latter task. Because the 5V logic output from the D-latches is insufficient to activate a solenoid alone, it instead controls solid state relays (SSR) which bridge the solenoids and a high voltage, high current power circuit. Additional circuitry is implemented to enable a "spike & hold" power scheme that increases efficiency and extends the solenoids' lifetime. Notably, every major IC component on the PCB is socketed to allow for easy user replacement; a production version of this hardware would likely replace SSRs with

transistors and directly solder all components to reduce overall costs. Nonetheless, the entire board can be built as specified for under $400.

The “one touch” analysis system is built around the same hardware detailed above, but is simply housed in a more presentable plastic enclosure (Figure 5.5.2a). This box also incorporates two of its own pressure regulators, so that only a single compressed air source is required for operation. Three grooves machined into the enclosure lid accommodate microfluidic chips, each with an accompanying pinless pressure manifold which is directly connected via Tygon tubing to the solenoids below. The manifolds are secured onto chips using four spring-loaded pins that latch into the enclosure lid. Each manifold is fabricated from a laser-cut acrylic plate which is further modified with o-ring grooves on the bottom/interface side, and 23-gage pins on the top side; modifications which allow for a stronger seal to the microfluidic chip and facile connection to the solenoids. A topside cover, also acrylic, creates a special channel which helps to lock down the removable blood reservoir manifold panel.

5.2.3 Microfluidics

Microfluidic chips for the “one touch” blood chip were fabricated using standard two-layer PDMS protocols[8]. An aluminum stencil was used for the control layer to standardize the dimensions of each chip, as discussed earlier for barcode chips used on the robotics (Chapter 4). Reservoirs and pinholes were punched manually (Harris Unicore), although a molded solution would increase device yield by improving alignment to the pinless manifold.

5.3 Results and Discussion

Our software package represents the centerpiece of our automation efforts. The aim is to allow non-technical users an accessible, GUI-based method to program microfluidics chips, much the same as NI Labview facilitates standard programming, or as the Windows operating system advances the DOS platform. To this end, there are three main panels which help the user set up and run their chip:

The first panel allows for very basic setup. Here, the user loads an image of their fluidic circuit into the main window and specifies the number of solenoids they will require to run their chip. A linear array of solenoids then appears schematically, and can be dragged anywhere onto the circuit image, e.g. atop a fluidic input or over a control valve. In this way, a very direct association is made between each solenoid and its function, a visual context that is completely lost when using manual flip switches or when programming simple Matlab/Labview scripts.

The second panel, or “State Panel”, optionally allows a user to create a set of pre-defined solenoid configurations. The concept of automation is predicated on the idea that a microfluidic circuit generally utilizes a limited number of discrete solenoid configurations (states) during normal operation. For example, there may be one configuration for priming reagents, a second for flowing them through the reaction chamber, etc. The State Panel facilitates creation of an arbitrary number of these states, which can then serve as the basis for an automation routine or simply as a shortcut to configure many solenoids in a single click during manual operation. States are created by clicking the “+” button in the State Panel, providing a label, and then toggling each of the solenoids to their desired state in the main

window. Solenoids may assume one of three configurations: on, off, or No Change, a state that specifies a solenoid will maintain its previous value. The NC state allows for greater flexibility and a reduced number of states when only one domain of a fluidic circuit requires action; the user is spared from adding all the permutations of configurations for multiple domains.

The third panel, or "Program Panel", is used to actually control the microfluidic chip and is the only panel which actually communicates with the attached hardware (Figure 5.5.3). At its very simplest, the user can click on valves in the main window and thereby toggle them on their microfluidic chip. Any states that were defined in the previous panel are also carried over and can be used as shortcuts to configure multiple solenoids simultaneously. An additional dialog on the panel allows the user to create automation routines for chips that are used repeatedly or that require precise timing. To create such a routine, one simply builds a queue of states and assigns a run time to each; the software then sequentially configures the solenoids for each state automatically, holding each configuration for the specified time without any further user intervention. During an automated run, the user is free to toggle solenoids manually via the main window, to skip or repeat steps, and to pause/resume the run.

By creating a visual context for solenoid control, users are able to click directly on the part of their fluidic circuit that requires action. The intuitive nature of this interaction radically facilitates the control process, while the introduction of recorded states creates a convenient avenue for full-scale automation. The software is applicable to almost any continuous flow fluidic circuit, and we have found that designs of reasonable complexity can be automated with under half an hour's effort. As a test case, we used the software to optimize and automate the

aforementioned blood analysis chip. While the device reported in our original work required constant attention and manipulation over a five-hour period, our new version produced similar or better-quality data with just one hour of unattended run time, bringing the device firmly into the realm of practicality for the first time.

We now turn to the problem of commercial-scale automation, and mature the existing automation infrastructure into a true "one touch" system. While the aforementioned software solution is ideal for laboratory environments with trained users, it does require familiarity with microfluidic techniques, particularly as relates to chip setup, reagent loading, and run monitoring. With the development of our blood analysis chip (Chapter 2) and its impending use in clinical trials, we sought to create a portable, autonomous system that even untrained nurses or technicians could operate. To this end, we first addressed the microfluidic chip's reagent loading procedure. Traditionally, reagents are drawn into Tygon tubing via syringe, the tubing is affixed to the microfluidic chip, and it is then pressurized from behind; one must ensure that the fluid is drawn up in one continuous plug so as not to introduce air bubbles to the microchannels. We replaced this apparatus by creating on-chip macroscale reservoirs for each of the solutions; a 3mm-diameter reservoir can accommodate ca. 45µL of reagent, which is more than sufficient for a single-use blood chip. Notably, these reservoirs are easily filled using standard micropipettors, and may even be pre-filled in the lab if the chip is refrigerated thereafter. In practical use, we found the hydrophobic PDMS walls would sometimes induce air bubble formation within the reservoir while pipetting. However, when pressurized, these bubbles did not enter microfluidic channels until the entire reservoir's solution was depleted;

they are nonetheless avoided by adding 0.01% TritonX detergent to the solutions prior to pipetting.

To drive solutions and manipulate control valves, a custom “pinless” pressure manifold was designed to be clamped onto the topside of each chip (Figure 5.2b). The manifold is hardwired to appropriate solenoids via Tygon tubing from the topside, and contains through-holes to transmit pressure to the underside wherever a reservoir or control channel is located. Embedded o-rings around each through-hole interface provide some positional tolerance and also improve the seal strength; we were able to transmit up to 15psi to close control channels with only minor leakage. Alignment is accomplished by two posts which protrude from diagonal corners of the microfluidic chip; these mate with corresponding holes on the manifold and thereby provide registration for all the pressure interfaces in between. While generally convenient, the manifold approach imposes a limitation in that once a run is started, it cannot be removed to access the chip underneath without releasing activated control valves. This presented a problem for our blood chip, as the patient sample is preferably loaded immediately prior to its use near the middle of the experiment. In order to mitigate this, a small section of the manifold corresponding to the blood reservoir was excised and left to float freely. At the same time, a cover was added to the manifold that bears a slot immediately above the blood reservoir. Upon pipetting the patient sample into its reservoir, the free-floating manifold is positioned and a key is wedged into the cover slot, pressing the floating manifold into place so it can pressurize the reservoir. In practice, this solution proved quite robust, with only a small percentage of chip failures attributed to a poor seal over the blood reservoir. Thus, the hard-

wired pressure manifold, while specific to a single chip design, enables rapid, foolproof chip setup in one step without requiring technical knowledge of the underlying microfluidics.

As a final step towards commercial-scale automation for our blood chip, we created a new software package that simultaneously increases device throughput and presents a completely non-technical user interface – the eponymous "one touch" interface. Throughput is particularly important in the case of clinical trials where patient sampling may occur at irregular and/or tightly spaced intervals dictated by doctors' schedules, which can be problematic if each test requires one uninterrupted hour to complete. Our original software was designed to automate a single chip at a time; attempts to run multiple chips simultaneously required that they also run synchronously – an obvious limitation for experiments requiring fresh blood samples given the aforementioned scheduling realities. Thus, we created a companion package to our initial automation software that allows users to import solenoid setups, states, and automation routines generated in the latter, and execute them in a parallel but asynchronous manner. The key step is a dialog which allows one to map multiple instances of an automated chip across the 64-solenoid array so that each instance acquires its own dedicated solenoids. The underlying run logic is restructured to allow each such instance to proceed simultaneously and asynchronously. At the same time, the run details are entirely obscured from the end user: the main interface simply consists of three buttons which, when pressed, initiate the automation sequence for a particular instance (Figure 5.5.4). A pair of status bars track the overall run progress and the current step's progress for the operator, but otherwise no further interaction is required.

The combined result of our chip redesign, pressure manifold interface, and refocused automation software is an integrated system that allows non-technical users to rapidly and conveniently operate sophisticated microfluidic devices with little training and only standard laboratory pipetting techniques. In practice, we found that a typical blood chip experiment could be set up and started in less than five minutes, and subsequently required only brief intervention when adding a fresh blood sample. The resulting data quality was indistinguishable from a manually-operated microfluidics run, and the overall device failure rate across dozens of runs was under 10%. We also assayed the degree of chip-to-chip reproducibility achieved in our automated assays by sequentially running nine blood chips; a cocktail of recombinant proteins was substituted for blood samples as our analyte, and Figure 5.5.5 shows the resulting coefficient of variation (CV) for two of these, as measured across the chips. Both IL-6 and CRP were detected with under 30% CV. Although our targeted reproducibility was 10%, it is likely that the figure we achieved is sufficient for blood work, where perturbations generally result in fold changes rather than small percent changes. Regrettably, we have no metric with which to measure the improvements in consistency over manually performed experiments, as no such testing with the latter has been performed.

5.4 Conclusions

We have presented here a set of software and hardware tools that are ideal for streamlining and automating microfluidic operations both at the laboratory scale and at small commercial scales. In addition to the immediate benefit of convenience, automation routines help to

eliminate operator-derived inconsistencies from one experiment to the next. Likewise, they can also aid in translating protocols and results across multiple research labs. As the microfluidics field transitions from basic component development to the application of integrated, mature fluidic systems to discovering novel science, these factors will continue to grow increasingly important.

5.5 Figures

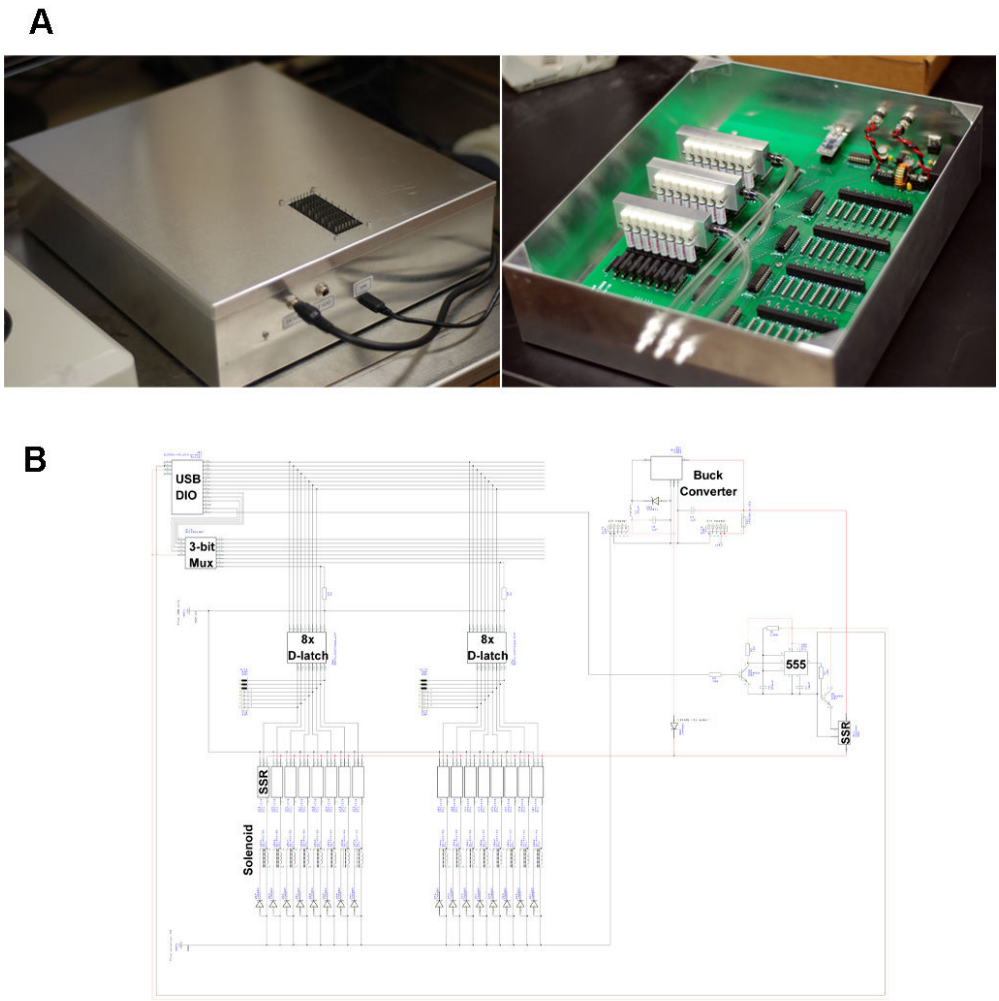


Figure 5.5.1 (A) The solenoid control hardware for laboratory-scale use features USB and power inputs and has a small panel at the top with a 64-pin interface for use with microfluidics chips. The internals are designed to be modular, so that solenoids can simply be plugged in as needed; here, 24 solenoids are in use. **(B)** Partial schematic of the solenoid electronics, which depicts how the USB chip's 14 DIO lines are multiplexed to control 64 solenoids.

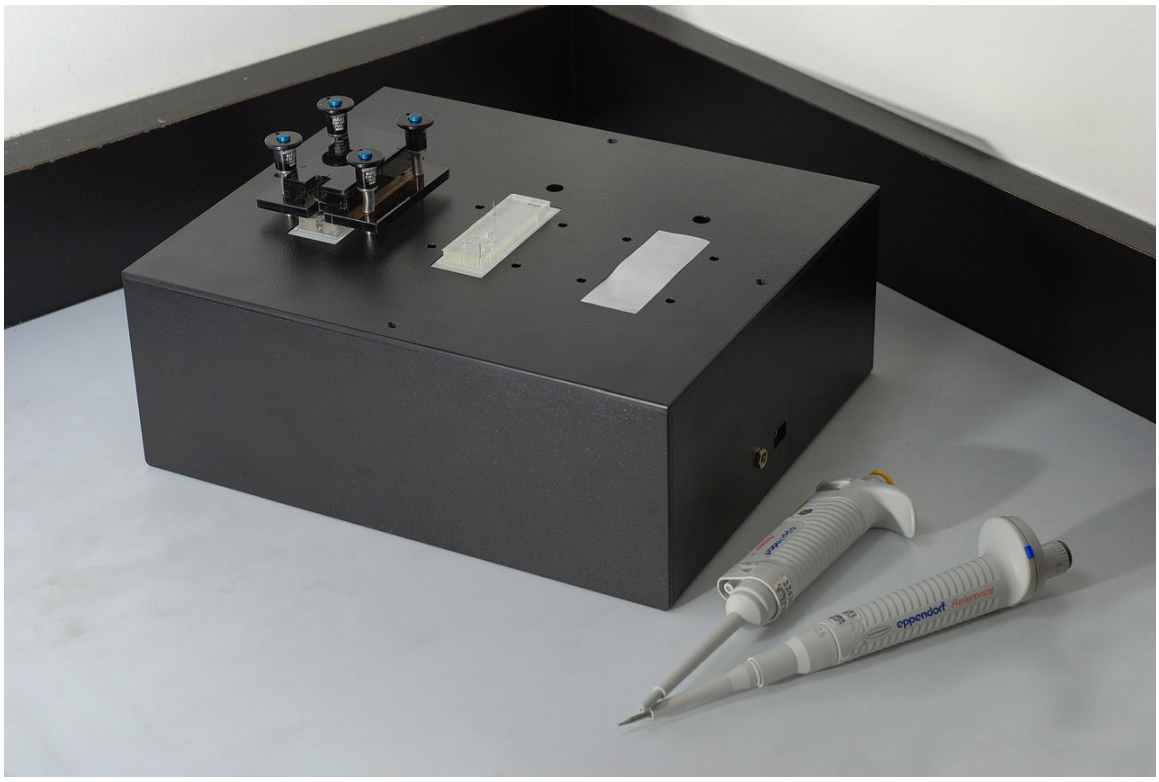


Figure 5.5.2 The fully automated blood chip apparatus features minimal inputs and 3 slots on its lid to accommodate chips. Here, only one pressure manifold is shown for clarity

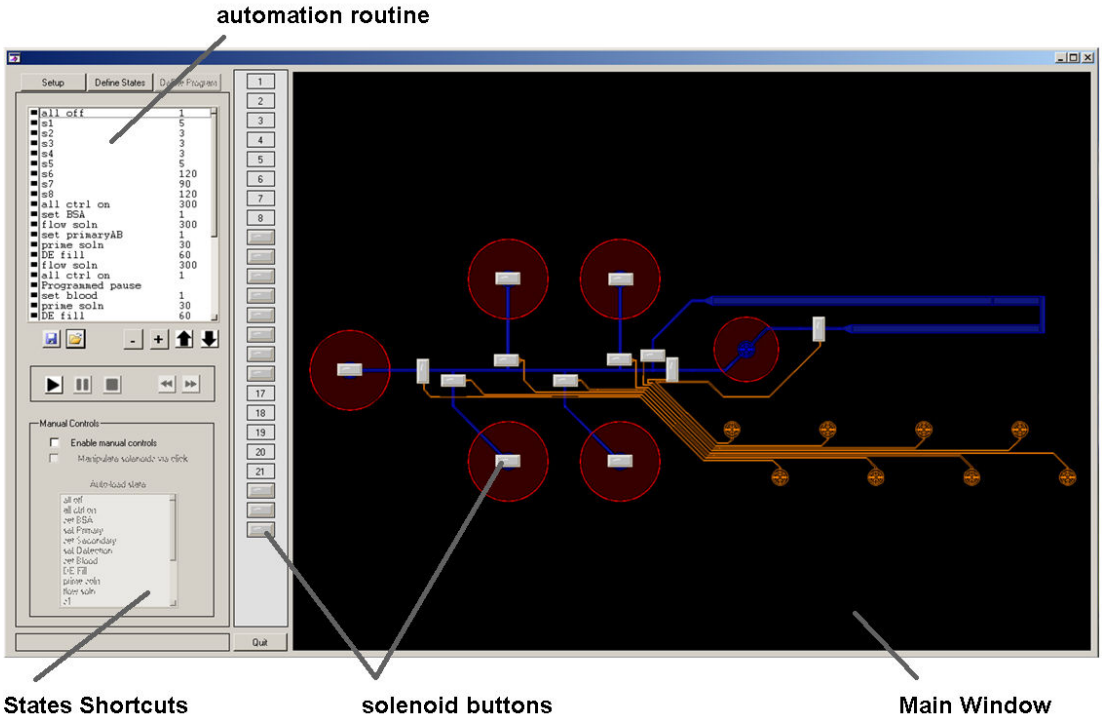


Figure 5.5.3 The program control window of our microfluidic control and automation software. Program and states panels are labeled; the microfluidic circuit image is overlaid with buttons representing solenoids. These may be manually clicked to toggle them, or they can be configured simultaneously by states set up in the states panel.

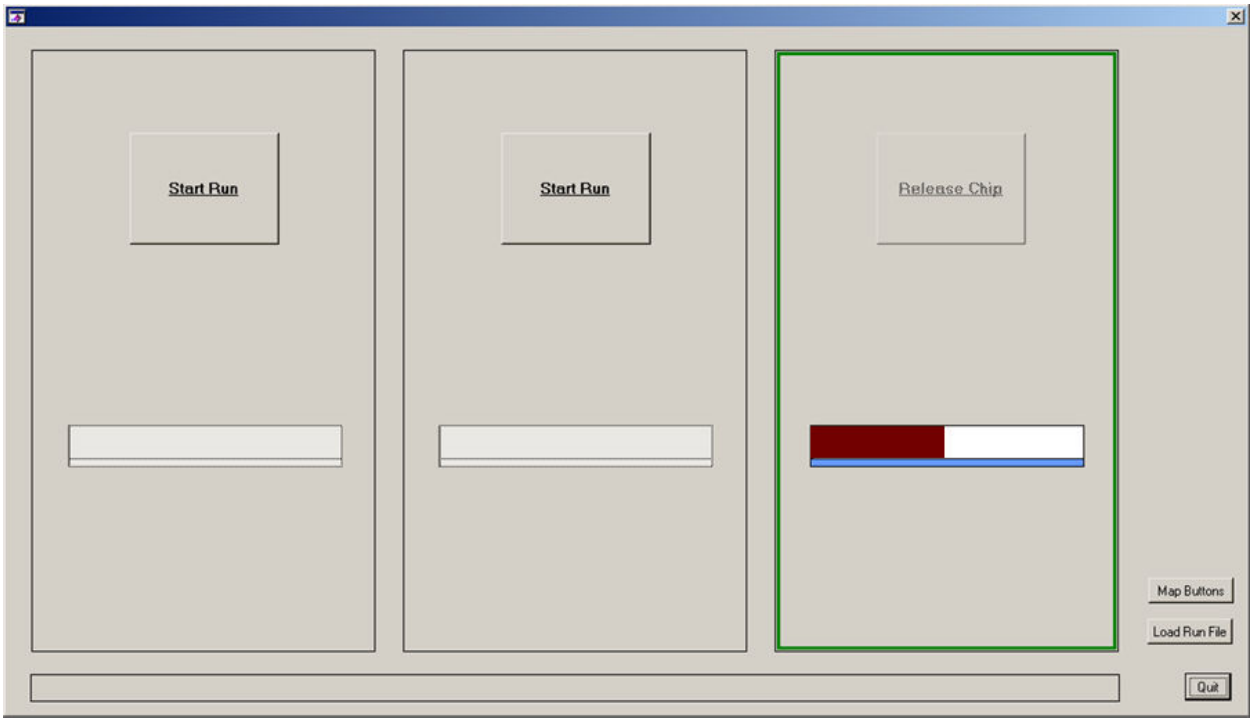


Figure 5.5.4 The fully autonomous blood chip utilizes an extremely simplified interface. Three buttons correspond to the slots on the control box lid, and allow the user to start any one of the three experiments. Red and blue bars indicate progress of the entire run and the current step, respectively.

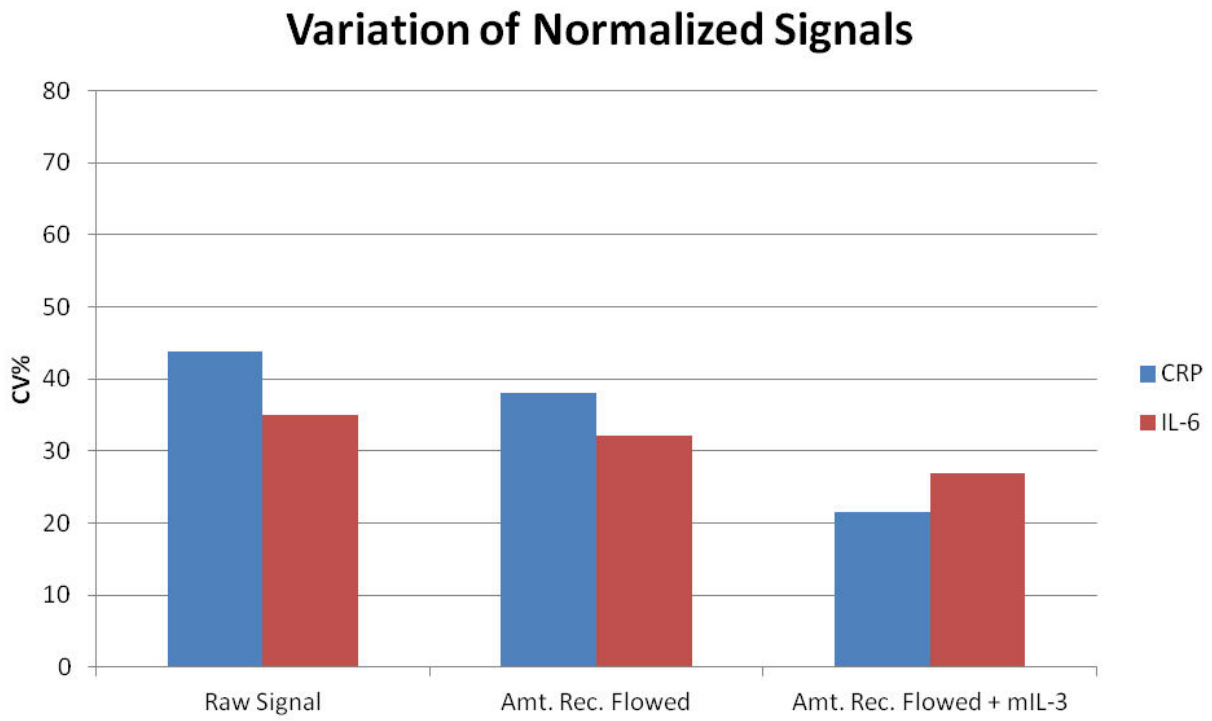


Figure 5.5.5. Coefficient of variation of CRP and IL-6 signals across 9 blood chips.

5.6 References

1 Becker, H. & Gärtner, C. Polymer microfabrication technologies for microfluidic systems. *Analytical and Bioanalytical Chemistry* **390**, 89-111-111, (2008).

2 Haber, C. Microfluidics in commercial applications; an industry perspective. *Lab on a Chip* **6**, 1118-1121, (2006).

3 Pollack, M. G., Fair, R. B. & Shenderov, A. D. Electrowetting-based actuation of liquid droplets for microfluidic applications. *Applied Physics Letters* **77**, 1725-1726, (2000).

4 Schwartz, J. A., Vykoukal, J. V. & Gascoyne, P. R. C. Droplet-based chemistry on a programmable micro-chip. *Lab on a Chip* **4**, 11-17, (2004).

5 Chakrabarty, K. Design Automation and Test Solutions for Digital Microfluidic Biochips. *Circuits and Systems I: Regular Papers, IEEE Transactions on* **57**, 4-17, (2010).

6 Gascoyne, P. R. C. *et al.* Dielectrophoresis-based programmable fluidic processors. *Lab on a Chip* **4**, 299-309, (2004).

7 Shih, S. C. C., Fobel, R., Kumar, P. & Wheeler, A. R. A feedback control system for high-fidelity digital microfluidics. *Lab on a Chip* **11**, 535-540, (2011).

8 Unger, M. A., Chou, H.-P., Thorsen, T., Scherer, A. & Quake, S. R. Monolithic Microfabricated Valves and Pumps by Multilayer Soft Lithography. *Science* **288**, 113-116, (2000).

9 Leslie, D. C. *et al.* Frequency-specific flow control in microfluidic circuits with passive elastomeric features. *Nat Phys* **5**, 231-235, (2009).

10 Mosadegh, B. *et al.* Integrated elastomeric components for autonomous regulation of sequential and oscillatory flow switching in microfluidic devices. *Nat Phys* **6**, 433-437, (2010).

11 Grover, W. H., Ivester, R. H. C., Jensen, E. C. & Mathies, R. A. Development and multiplexed control of latching pneumatic valves using microfluidic logical structures. *Lab on a Chip* **6**, 623-631, (2006).

12 Fordyce, P. M. *et al.* De novo identification and biophysical characterization of transcription-factor binding sites with microfluidic affinity analysis. *Nat Biotech* **28**, 970-975, (2010).

13 Fan, H. C., Wang, J., Potanina, A. & Quake, S. R. Whole-genome molecular haplotyping of single cells. *Nat Biotech* **29**, 51-57, (2011).

14 Gu, W., Zhu, X., Futai, N., Cho, B. S. & Takayama, S. Computerized microfluidic cell culture using elastomeric channels and Braille displays. *Proceedings of the National Academy of Sciences of the United States of America* **101**, 15861-15866, (2004).

15 Thies, W., Urbanski, J., Thorsen, T. & Amarasinghe, S. Abstraction layers for scalable microfluidic biocomputing. *Natural Computing* **7**, 255-275-275, (2008).

16 Amin, A. M., Thottethodi, M., Vijaykumar, T. N., Wereley, S. & Jacobson, S. C. Aquacore: a programmable architecture for microfluidics. *SIGARCH Comput. Archit. News* **35**, 254-265, (2007).

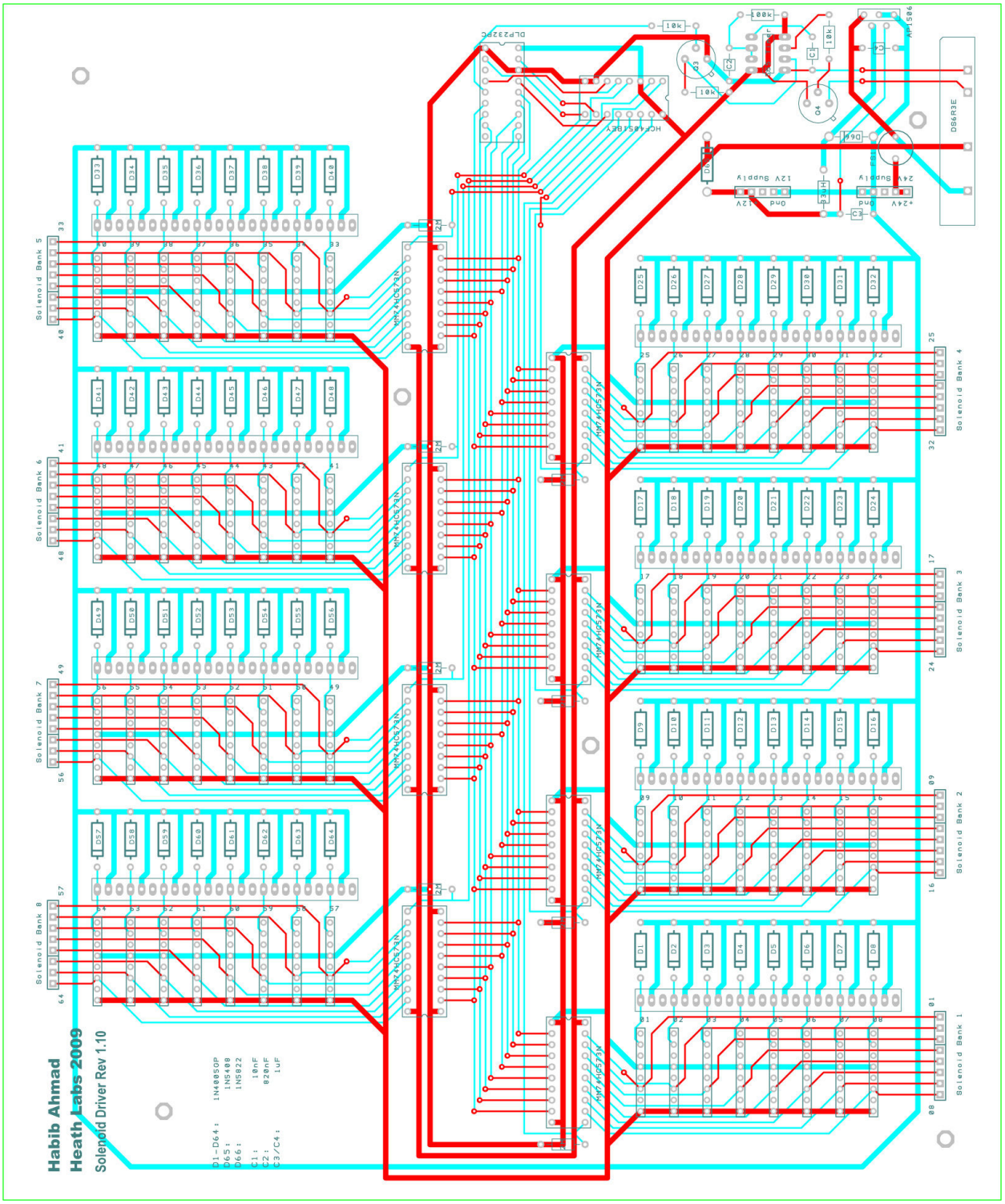
17 Urbanski, J. P., Thies, W., Rhodes, C., Amarasinghe, S. & Thorsen, T. Digital microfluidics using soft lithography. *Lab on a Chip* **6**, 96-104, (2006).

18 Harris, G., Montgomery, J., Lee, M. & Worthington, G. Object oriented microfluidic design method and system. USA patent US 2005/0149304 A1 (2005).

19 Hong-Dun, L. *et al.* in *Nuclear Science Symposium Conference Record, 2006. IEEE.* 2095-2098.

20 Kunin, W., Keshavarzian, N. & Wang, B. AutoCAD Add-on for Simplified Design of Microfluidic Devices. *Journal of Computer Chemistry, Japan* **9**, 183-196, (2010).

5.7 Appendix A: PCB Design



5.8 Appendix B: Source Code

The following pages contain the Labwindows CVI source code used to control the solenoid hardware for both laboratory use and the specialized “one touch” analysis device.

```
1   #include   <rs232.h>
2   #include   <easyio.h>
3   #include   <utility.h>
4   #include   <formatio.h>
5   #include   <ansi_c.h>
6   #include   "buttonSetup.h"
7   #include   <cvirte.h>
8   #include   <userint.h>
9   #include   "mainPanel.h"
10  #include   "EditState.h"
11
12  #define   MAXSOLENOIDS 64
13  #define   MAXSTATES 100
14  #define   MAXSTEPS 100
15  #define   PAUSECODE -5
16
17  static   int  pnlMain;
18  static   int  pnlButtonSetup;
19  static   int  pnlSetStates;
20  static   int  pnlSetProgram;
21  static   int  pnlEditStep;
22
23  char imageFileName[MAX_PATHNAME_LEN];
                                          //global to keep track for
    "save" function
24
25  int  usbPort;
26  int  clickMode;
27  int  activeControl, activeColor, ncColor = VAL_YELLOW;
28  int  pictFrameX, pictFrameY;
29  int  numSolenoids;

30  int goStatus;
                                                                    //-1:
    Stop     0: Pause       1: Resume Run       2: Running normally
31  int stepOverride;


32
33  int solenoids[MAXSOLENOIDS][7];
                                            //[0] Button ID     [1] X
    pos    [2] Y pos     [3] placeholder ID     [4] placeholder txt [5]
    origX    [6] origY
34  int solenoidSize[5] = {10, 13, 16, 19, 22};
                                  //width of solenoid buttons
35  int solenoidState[MAXSTATES][MAXSOLENOIDS];
                                  //solenoid state for each program step
36  int uID[MAXSTATES];
```

```
                                                                    //unique ID
    for each state
37  int* stateList[MAXSTATES];
                                                          //correlate list to
    solenoidState[]
38  int* currState = 0;
                                                                    //ptr to
    current program step
39  int stepRef[MAXSTEPS][3];
                                                           //[0] state uID,
    [1] reference to stateList [2] duration for each step
40  int solenoidConfig[MAXSOLENOIDS];
                                                 //current configuration of
    all solenoids. Need this information for "Ignore" solenoids
41
42  int bbID = -1,
43      bbLeft = 291,
44      bbTop = 5,

45      bbX = 70,
46      bbY = 715;

        //bounding box for solenoids & placeholders
47
48
49  /****** DLP232 commands ******/
50  int inhibitOn = 112, inhibitOff = 113, pulseVon = 116, pulseVoff =
    38;
51  int busLines[8][2] = {81, 49, 87, 50, 69, 51, 82, 52, 84, 53, 89,
    54, 85, 55, 73, 56};          //array[bus line][off/on]
52  int ctrlLines[3][2] = {105, 104, 101, 100, 97, 47};
53  int chipID[8][3] = {0,0,0, 0,0,1, 0,1,0, 0,1,1, 1,0,0, 1,0,1, 1,1,0
    , 1,1,1};
54  /*****************************/
55
56
57  char pauseActive[100] = "imgs/PauseActive.pcx";
58  char pauseNormal[100] = "imgs/Pause.pcx";
59
60
61
62
63
64
65
66  void setStatus(char* msg)
67  {
```

```
68          SetCtrlVal(mainPanel, mainPanel_txtStatus, msg);
69          return;
70      }
71
72
73      int setSolenoid(int solenoidNum, int state)
74      {
75          int bank = 0;
76          int i, offset;
77
78
79          bank = (int)(solenoidNum/8);
                                                        //work out port &
            solenoid values
80          solenoidNum = solenoidNum % 8;
81          offset = bank*8;
82
83          for(i = 0; i < 3; i++) ComWrtByte (usbPort, ctrlLines[i][chipID
            [bank][i]]);                //select chip
84          for(i = 0; i < 8; i++) ComWrtByte (usbPort, busLines[i][
            solenoidConfig[offset+i]]);         //restore prior configuration
            to bus
85
86          ComWrtByte (usbPort, busLines[solenoidNum][state]);
                            //write new value to bus
87          solenoidConfig[offset + solenoidNum] = state;
                                  //and record to memory
88
89          ComWrtByte (usbPort, inhibitOff);
                                                  //latch values
90          Delay(0.05);
            //Delay(0.005);

91          ComWrtByte (usbPort, inhibitOn);
92          ComWrtByte (usbPort, pulseVon);
                                                    //switch values
93          Delay(0.05);            //Delay(0.001);
94          ComWrtByte (usbPort, pulseVoff);
95
96
97          return 0;
98      }
99
100 101
102     int setState(int indexNo)
103     {
```

```
104         int i, j, numBanks, offset;
105
106         numBanks = (int)(numSolenoids/8);
                                                //determine how many banks
            are in use
107         if(numSolenoids == MAXSOLENOIDS) numBanks--;
                                    //    maximum solenoids gives faulty
            value for numBanks
108         for(i = 0; i <= numBanks; i++)
                                                  //for each bank
109         {    offset = 8*i;
110              for(j = 0; j < 3; j++) ComWrtByte (usbPort, ctrlLines[j][
                 chipID[i][j]]);                // select chip
111              for(j = 0; j < 8; j++)

                      // setup bus
112              {    if(solenoidState[indexNo][offset+j] == 2) ComWrtByte (
                 usbPort, busLines[j][solenoidConfig[offset+j]]); //if
                 ignore, retrieve previous value
113                   else
114                   {    ComWrtByte (usbPort, busLines[j][solenoidState[
                      indexNo][offset+j]]);            //     else write new value
115                        solenoidConfig[offset+j] = solenoidState[indexNo][
                           offset+j];                    //     and record to memory
116                   }
117              }
118
119              ComWrtByte (usbPort, inhibitOff);
                                                //   latch values
120              Delay(0.05);
                 //Delay(0.005);

121              ComWrtByte (usbPort, inhibitOn);
122         }
123
124         ComWrtByte (usbPort, pulseVon);
                                                  //switch values
125         Delay(0.05);               //Delay(0.001);
126         ComWrtByte (usbPort, pulseVoff);
127
128
129         return 0;
130    }
131
132
133    void initSolenoids()
134    {
```

```
135         int i;
136
137         for(i = 0; i < 64; i++)
138             setSolenoid(i, 0);
139
140         return;
141     }
142
143
144     int activeSolenoid()
145     {
146         int i = 0;
147
148         while((activeControl != solenoids[i][0]) && (++i < numSolenoids
            -1));
149
150         return i;
151     }
152
153
154     int openUSB()
155     {
156         int i, success, response;
157         int comPort = -1;
158         int ping = 39;
                                                                    //ascii
            code for apostrophe (')
159
160
161         for(i = 3; i < 4; i++)
162     // for(i = 0; i < 10;
        i++)                                                //open com ports
        sequentially
163         {   success = OpenComConfig (i, "", 460800, 0, 8, 1, 512, 512);
                 // open port
164             if(success == 0)
                                                                    // if
                successfull
165             {   while(ComWrtByte (i, ping) != 1);
                                                //     issue ping
166                 response = ComRdByte (i);
                                                        //     read response
167                 if(response == 'Q')
                                                                    //     if ping
                    response from DLP232
168                 {   comPort = i;
                                                                    //
```

```
                record port number
169                 i = 10;

                    //      exit loop
170             }
171             CloseCom (i);
                                                                    //
                else close com port
172         }
173     }
174
175     return comPort;
176 }
177
178
179 void setup()
180 {
181     int i, j;
182
183     GetCtrlAttribute (mainPanel, mainPanel_pictScheme, ATTR_LEFT, &
        pictFrameX);
184     GetCtrlAttribute (mainPanel, mainPanel_pictScheme, ATTR_TOP, &
        pictFrameY);
185
186     for(i = 0; i < MAXSOLENOIDS; i++)
187         for (j = 0; j < 5; j++) solenoids[i][j] = -1;
188
189     bbID = NewCtrl (mainPanel, CTRL_FLAT_BOX, "", bbTop, bbLeft);
190     SetCtrlAttribute (mainPanel, bbID, ATTR_HEIGHT, bbY);
191     SetCtrlAttribute (mainPanel, bbID, ATTR_WIDTH, bbX);
192     SetCtrlAttribute (mainPanel, bbID, ATTR_FRAME_COLOR,
        VAL_OFFWHITE);
193     SetCtrlAttribute (mainPanel, bbID, ATTR_ZPLANE_POSITION, 100);
194
195     activeColor = VAL_RED;
196     SetCtrlAttribute (pnlSetStates, pnlStates_numStates,
        ATTR_MAX_VALUE, MAXSTATES);
197
198     for(i = 0; i < MAXSTATES; i++)
199     {   solenoidState[i][0] = -1;
                                                //-1 indicates open slot
200         stateList[i] = NULL;
201         uID[i] = -1;
202     }
203
204     clickMode = 0;
205
```

```
206
207         setStatus("Finding USB device");
208         ProcessDrawEvents();
209         usbPort = openUSB();
                                                                    //find DLP232PC
210         if(usbPort == -1)
211         {    MessagePopup("Error", "The solenoid device could not be
            located; run functions will be disabled." );         //if not found,
            give msg
212              return;
213         }
214         else

            //otherwise
215         {    SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdPlay,
            ATTR_DIMMED, 0);               // enable run controls
216              SetCtrlAttribute (pnlSetProgram,
                 pnlProgram_chkEnableManualCtrl , ATTR_DIMMED, 0);
217              ComWrtByte (usbPort, 117);

                 //'u' = no analog channels
218              ComWrtByte (usbPort, inhibitOn);

219         }
220
221
222         initSolenoids();
223         setStatus("");
224
225
226         return;
227    }
228
229 230
231    int main (int argc, char *argv[])
232    {
233         if (InitCVIRTE (0, argv, 0) == 0)
234              return -1; /* out of memory */
235         if ((pnlMain = LoadPanel (0, "mainPanel.uir", mainPanel)) < 0)
236              return -1;
237         if ((pnlButtonSetup = LoadPanel (0, "buttonSetup.uir", btnPanel
            )) < 0)
238              return -1;
239         if ((pnlSetStates = LoadPanel (pnlMain, "mainPanel.uir",
            pnlStates)) < 0)
240              return -1;
```

```
241         if ((pnlSetProgram = LoadPanel (pnlMain, "mainPanel.uir",
            pnlProgram)) < 0)
242             return -1;
243         if ((pnlEditStep = LoadPanel (pnlMain, "EditState.uir",
            pnlEdtStep)) < 0)
244             return -1;
245
246
247
248
249     //   setup();
250
251         DisplayPanel (pnlMain);
252         setup();
253         RunUserInterface ();
254         DiscardPanel (pnlMain);
255         return 0;
256     }
257
258     int CVICALLBACK buttonPress (int panel, int control, int event,
259             void *callbackData, int eventData1, int eventData2)
260     {
261         int xDim, yDim, active, index, mode, i;
262
263
264
265         if(clickMode == 0)
                                                                    //if setup
            mode
266         {
267             switch (event)
268             {
269                 case EVENT_GOT_FOCUS:
270                     activeControl = control;
271                     index = activeSolenoid();
272
273                     SetCtrlVal (panel, control, 1);
274                     SetCtrlAttribute (mainPanel, solenoids[index][3],
                        ATTR_FRAME_COLOR, activeColor);
275                     SetCtrlAttribute (mainPanel, solenoids[index][4],
                        ATTR_TEXT_BGCOLOR, activeColor);
276                     break;
277
278                 case EVENT_LOST_FOCUS:
279                     SetCtrlVal (panel, control, 0);
280                     index = activeSolenoid();
281
```

```
282                     SetCtrlAttribute (mainPanel, solenoids[index][3],
                        ATTR_FRAME_COLOR, VAL_TRANSPARENT);
283                     SetCtrlAttribute (mainPanel, solenoids[index][4],
                        ATTR_TEXT_BGCOLOR, VAL_TRANSPARENT);
284                     break;
285
286                 case EVENT_COMMIT:
287                     GetCtrlVal(panel, control, &active);
288                     if(!active) SetCtrlVal(panel, control, 1);
289                     break;
290
291                 case EVENT_LEFT_CLICK:
292
293                     break;
294
295                 case EVENT_RIGHT_CLICK:
296                     GetCtrlVal(panel, control, &active);
297                     if(!active) break;
                                                              //only
                        rotate active button
298
299                     GetCtrlAttribute (panel, control, ATTR_HEIGHT, &
                        yDim);
300                     GetCtrlAttribute (panel, control, ATTR_WIDTH, &xDim
                        );
301
302                     SetCtrlAttribute (panel, control, ATTR_HEIGHT, xDim
                        );
303                     SetCtrlAttribute (panel, control, ATTR_WIDTH, yDim);
304                     SetCtrlAttribute (mainPanel, control,
                        ATTR_LABEL_WIDTH, yDim);
305                     break;
306
307                 case EVENT_RIGHT_DOUBLE_CLICK:
308                     GetCtrlVal(panel, control, &active);

309                     if(!active) break;
310                     index = activeSolenoid();
311
312                     SetCtrlAttribute (mainPanel, control, ATTR_LEFT,
                        solenoids[index][5]);
313                     SetCtrlAttribute (mainPanel, control, ATTR_TOP,
                        solenoids[index][6]);
314
315                     GetCtrlAttribute (panel, control, ATTR_HEIGHT, &
                        yDim);
316                     GetCtrlAttribute (panel, control, ATTR_WIDTH, &xDim
```

```
                    );
317                 if(yDim > xDim)
                                                                //if
                    rotated, return to upright
318                 {   SetCtrlAttribute (panel, control, ATTR_HEIGHT,
                    xDim);
319                     SetCtrlAttribute (panel, control, ATTR_WIDTH,
                        yDim);
320                     SetCtrlAttribute (mainPanel, control,
                        ATTR_LABEL_WIDTH, yDim);
321                 }
322
323                 solenoids[index][1] = -1; 324
                    solenoids[index][2] = -1;
325                 break;
326
327             }
328         }
329
330         if(clickMode == 1)
                                                              //if state
            mode
331         {   switch (event)
332             {   case EVENT_COMMIT:
333                     activeControl = control;
334                     index = activeSolenoid();
335                     GetCtrlVal(panel, control, &active);
                                            // get value
336                     if(!active && (currState[index] == 1))

337                     {   SetCtrlAttribute (mainPanel, solenoids[index][3
                        ], ATTR_FRAME_COLOR, ncColor);
338                         SetCtrlAttribute (mainPanel, solenoids[index][4
                        ], ATTR_TEXT_BGCOLOR, ncColor);
339                         SetCtrlAttribute (mainPanel, solenoids[index][0
                        ], ATTR_ON_COLOR, ncColor);
340                         SetCtrlVal(mainPanel, solenoids[index][0], 2);
341                         currState[index] = 2;
342                         break;
343                     }
344                     if(active)
                                                                      //
                        & toggle
345                     {   SetCtrlAttribute (mainPanel, solenoids[index][3
                        ], ATTR_FRAME_COLOR, activeColor);
346                         SetCtrlAttribute (mainPanel, solenoids[index][4
                        ], ATTR_TEXT_BGCOLOR, activeColor);
```

```
347                         SetCtrlAttribute (mainPanel, solenoids[index][0
                            ], ATTR_ON_COLOR, activeColor);
348                         currState[index] = 1;
349                     }


350                     else
351                     {   SetCtrlAttribute (mainPanel, solenoids[index][3
                        ], ATTR_FRAME_COLOR, VAL_TRANSPARENT);
352                         SetCtrlAttribute (mainPanel, solenoids[index][4
                            ], ATTR_TEXT_BGCOLOR, VAL_TRANSPARENT);
353                         currState[index] = 0;
354                     }
355                     break;
356             }
357         }
358
359
360         if(clickMode == 2)
                                                                  //if program
            mode
361         {   switch (event)
362             {   case EVENT_COMMIT:
363                     GetCtrlVal(panel, control, &active);
                                                  // get value
364                     SetCtrlVal(panel, control, !active);
                                                  // undo click
365                     break;
366             }


367         }
368
369         if(clickMode == 3)
                                                                  //if program
            mode w/solenoid toggle
370         {   switch (event)
371             {   case EVENT_COMMIT:
372                     activeControl = control;
373                     index = activeSolenoid();
374                     GetCtrlVal(panel, control, &active);
                                                  // get value
375                     if(active)
                                                                        //
                        & toggle
376                     {   SetCtrlAttribute (mainPanel, solenoids[index][3
                        ], ATTR_FRAME_COLOR, activeColor);
```

```
377                         SetCtrlAttribute (mainPanel, solenoids[index][4
                            ], ATTR_TEXT_BGCOLOR, activeColor);
378                         setSolenoid(index, 1);
379                     }


380                     else
381                     {   SetCtrlAttribute (mainPanel, solenoids[index][3
                        ], ATTR_FRAME_COLOR, VAL_TRANSPARENT);
382                         SetCtrlAttribute (mainPanel, solenoids[index][4
                            ], ATTR_TEXT_BGCOLOR, VAL_TRANSPARENT);
383                         setSolenoid(index, 0);
384                     }
385                     break;
386             }
387         }
388
389         return 0;
390     }
391
392     int CVICALLBACK quitProgram (int panel, int control, int event,
393             void *callbackData, int eventData1, int eventData2)
394     {
395         if(event != EVENT_COMMIT) return 0;
396
397         QuitUserInterface(0);
398
399         return 0;
400     }
401
402     int CVICALLBACK setButtons (int panel, int control, int event,
403             void *callbackData, int eventData1, int eventData2)
404     {
405         int ctrlWidth, ctrlHeight, xPos, yPos, index;
406
407         if(clickMode != 0) return 0;
                                                        //no repositioning
            allowed outside setup mode
408
409
410         switch (event)
411             {
412             case EVENT_LEFT_DOUBLE_CLICK:
413                 GetCtrlAttribute(mainPanel, activeControl, ATTR_HEIGHT,
                     &ctrlHeight);
414                 GetCtrlAttribute(mainPanel, activeControl, ATTR_WIDTH,
                    &ctrlWidth);
```

```
415
416                 yPos = eventData1 - (int)(0.5 * ctrlHeight);
417                 xPos = eventData2 - (int)(0.5 * ctrlWidth);
418
419                 SetCtrlAttribute (mainPanel, activeControl, ATTR_LEFT,
                    xPos);
420                 SetCtrlAttribute (mainPanel, activeControl, ATTR_TOP,
                    yPos);
421
422                 index = activeSolenoid();
423                 solenoids[index][1] = xPos ;//- pictFrameX;
424                 solenoids[index][2] = yPos ;//- pictFrameY;
425                 break;
426             }
427         return 0;
428     }
429
430     int CVICALLBACK launchButtonSetup (int panel, int control, int
    event,
431
432             void *callbackData, int eventData1, int eventData2)
433     {
434         if(event != EVENT_COMMIT) return 0;
435
436             InstallPopup (pnlButtonSetup);
437
438
439         return 0;
440     }
441
442     int CVICALLBACK cancelNewButtons (int panel, int control, int event,
443             void *callbackData, int eventData1, int eventData2)
444     {
445         if(event != EVENT_COMMIT) return 0;
446
447             RemovePopup (pnlButtonSetup);
448
449         return 0;
450     }
451
452     void DestroyArray(int mode)
                                                        //mode: 1 = full
    wipe, 0 = keep pos
453     {
454         int i = 0, j = 0;
```

```
455
456         while((i < MAXSOLENOIDS) && (solenoids[i][0] != -1))
457         {   DiscardCtrl(mainPanel, solenoids[i][0]);
                                        //discard button
458             DiscardCtrl(mainPanel, solenoids[i][3]);
                                            //and placeholder
459             DiscardCtrl(mainPanel, solenoids[i][4]);
460
461             solenoids[i][0] = -1; 462
                solenoids[i][3] = -1; 463
                solenoids[i][4] = -1;
464
465             if(mode)
466             {   solenoids[i][1] = -1; 467
                solenoids[i][2] = -1;
468             }
469             i++;
470         }
471
472         return;
473     }
474
475
476     int CreateButtonArray(int arrSize, int bSize)
477     {
478         int i, j;
479         int modifier = 0;//bSize, arrSize;
480         char lbl[10];
481         int lblX, lblY;
482
483
484         bSize = solenoidSize[bSize-1];
485
486         if(arrSize < 33) modifier = 0;
487         else modifier = -(bSize+4);
488
489         for(i = 0; i < arrSize; i++)
                                                        //for buttons needed
490         {   if(i == 32) modifier = -modifier;
                                                // switch column for after 32
491             if(solenoids[i][0] == -1)
                                                            // if no button
                exists
492             {   sprintf(lbl, "%i", i+1);
                                                        //     record solenoid
                num
493                 solenoids[i][3] = NewCtrl (mainPanel, CTRL_FLAT_BOX,
```

```
                lbl, (bbTop + 5 + ((i%32)*(bSize + 9))),            //
                create container
494                                     modifier + (bbLeft + (int)(
                                        0.5*(bbX-(2*(bSize + 2
                                        ))))));
495
496
497             SetCtrlAttribute (mainPanel, solenoids[i][3],
                ATTR_HEIGHT, bSize + 4);
498             SetCtrlAttribute (mainPanel, solenoids[i][3],
                ATTR_WIDTH, (2*bSize + 4));
499             SetCtrlAttribute (mainPanel, solenoids[i][3],
                ATTR_FRAME_COLOR, VAL_TRANSPARENT);
500             SetCtrlAttribute (mainPanel, solenoids[i][3],
                ATTR_ZPLANE_POSITION, 0);
501
502             solenoids[i][4] = NewCtrl (mainPanel, CTRL_TEXT_MSG,
                lbl, 0,0);            //     create label
503             GetCtrlAttribute (mainPanel, solenoids[i][4],
                ATTR_WIDTH, &lblX);
504             GetCtrlAttribute (mainPanel, solenoids[i][4],
                ATTR_HEIGHT, &lblY);
505
506             SetCtrlVal (mainPanel, solenoids[i][4], lbl);
507             SetCtrlAttribute (mainPanel, solenoids[i][4], ATTR_LEFT
                , modifier + (bbLeft + (int)(0.5*(bbX - lblX))));
508             SetCtrlAttribute (mainPanel, solenoids[i][4], ATTR_TOP,
                 (bbTop + 6 + ((i%32)*(bSize + 9)) + (int)(0.5*(bSize
                + 4 - lblY))));
509             SetCtrlAttribute (mainPanel, solenoids[i][4],
                ATTR_SIZE_TO_TEXT, 1);
510             SetCtrlAttribute (mainPanel, solenoids[i][4],
                ATTR_TEXT_BGCOLOR, VAL_TRANSPARENT);
511             SetCtrlAttribute (mainPanel, solenoids[i][4],
                ATTR_ZPLANE_POSITION, 0);
512
513
514             solenoids[i][0] = NewCtrl (mainPanel, "", CTRL_SQUARE_LED,
                (bbTop + 5 + ((i%32)*(bSize+9)) +         2),
515                                        modifier +    (bbLeft + (int)(
                                           0.5*(bbX -    2*bSize))));
                                                  //     create solenoid
516
517             SetCtrlAttribute (mainPanel, solenoids[i][0],
                ATTR_HEIGHT, bSize);
518             SetCtrlAttribute (mainPanel, solenoids[i][0],
                ATTR_WIDTH, (2*bSize));
```

```
519                 SetCtrlAttribute (mainPanel, solenoids[i][0],
                    ATTR_ZPLANE_POSITION, 0);
520                 SetCtrlAttribute (mainPanel, solenoids[i][0],
                    ATTR_CTRL_MODE, VAL_HOT);
521                 SetCtrlAttribute (mainPanel, solenoids[i][0],
                    ATTR_LABEL_HEIGHT, 3);
522                 SetCtrlAttribute (mainPanel, solenoids[i][0],
                    ATTR_LABEL_WIDTH, 2*bSize);
523                 SetCtrlAttribute (mainPanel, solenoids[i][0],
                    ATTR_LABEL_LEFT, VAL_RIGHT_ANCHOR);
524                 SetCtrlAttribute (mainPanel, solenoids[i][0],
                    ATTR_LABEL_TOP, (bbTop + 6 + (i*(bSize + 9)) + (int)(
                    0.5*(bSize + 4 - lblY))));
525                 SetCtrlAttribute (mainPanel, solenoids[i][0],
                    ATTR_LABEL_TEXT, " ");
526                 SetCtrlAttribute (mainPanel, solenoids[i][0],
                    ATTR_LABEL_BGCOLOR, VAL_TRANSPARENT);
527                 SetCtrlAttribute (mainPanel, solenoids[i][0],
                    ATTR_LABEL_VISIBLE, 0);
528                 InstallCtrlCallback (mainPanel, solenoids[i][0],
                    buttonPress, NULL);
529
530                 GetCtrlAttribute (mainPanel, solenoids[i][0], ATTR_LEFT
                    , &solenoids[i][5]);
531                 GetCtrlAttribute (mainPanel, solenoids[i][0], ATTR_TOP,
                     &solenoids[i][6]);
532             }
533
534             if(solenoids[i][1] != -1)

                // if prev. positioned
535             {   SetCtrlAttribute (mainPanel, solenoids[i][0], ATTR_LEFT
                , solenoids[i][1]);          //     recover position
536                 SetCtrlAttribute (mainPanel, solenoids[i][0], ATTR_TOP,
                     solenoids[i][2]);
537             }
538
539
540         }
541
542         for(i = arrSize; i < MAXSOLENOIDS; i++)
                                              //for remaining slots
543         {   if(solenoids[i][0] != -1)
                                                        // if button exists
544             {   DiscardCtrl(mainPanel, solenoids[i][0]);
                                         //     discard all associated
545                 DiscardCtrl(mainPanel, solenoids[i][3]);
```

```
546                 DiscardCtrl(mainPanel, solenoids[i][4]);
547
548                 for(j = 0; j < 5; j++) solenoids[i][j] = -1;
549             }
550         }
551
552          return 0;
553
554     }
555
556 557
558     void buildStatesRing()
559     {
560         char itemLabel[50];
561         int numStates, i;
562
563
564         ClearListCtrl(pnlEditStep, pnlEdtStep_rngStates);
                                    //remove prior list
565         ClearListCtrl(pnlSetProgram, pnlProgram_lstLoadState);
566         GetNumListItems(pnlSetStates, pnlStates_lstStates, &numStates);
                   //get total num states
567
568         InsertListItem (pnlEditStep, pnlEdtStep_rngStates, 0, "Pause
            for user input", -1);
569         for(i = 0; i < numStates; i++)
                                                                    //for each
            state
570         {    GetLabelFromIndex (pnlSetStates, pnlStates_lstStates, i,
            itemLabel);           // get label
571             InsertListItem (pnlEditStep, pnlEdtStep_rngStates, -1,
                itemLabel, i);
572             InsertListItem (pnlSetProgram, pnlProgram_lstLoadState, -1,
                 itemLabel, i);
573         }
574
575
576         return;
577     }
578
579 580
581     int refreshStepLinks()
582     {
583         int i, j, k, numSteps, numStates, status;
584         int errFlag = 0;
```

```
585         char itemLabel[50], newLabel[80];
586
587
588         GetNumListItems (pnlSetProgram, pnlProgram_lstProgSteps, &
            numSteps);
589         GetNumListItems (pnlSetStates, pnlStates_lstStates, &numStates);
590
591         for(i = 0; i < numSteps; i++)
592         {   if((stepRef[i][1] != PAUSECODE) && (stepRef[i][0] != uID[
            stepRef[i][1]]))                        //if state uIDs don't match
593             {   j = 0;
594                 while((stepRef[i][0] != uID[j]) && (j++ < numStates));
                                                        // go through
                    states & match uID
595                 if(j >= numStates) status = errFlag = PAUSECODE;
                                                                // if not
                    found, set flag to -1
596                 else status = j;

                                    //   else set to correct index
597
598
599                 for(j = i; j < numSteps; j++)

                        // proceed through remaining array
600                 {   if(stepRef[j][0] == stepRef[i][0])

                    //     if state is repeated
601                     {   if(status == PAUSECODE)

                                //     for status = not found
602                         {   CheckListItem (pnlSetProgram,
                            pnlProgram_lstProgSteps, j, 0);
                            //       uncheck list item
603                             GetLabelFromIndex (pnlSetProgram,
                            pnlProgram_lstProgSteps, j, itemLabel);
                            //       get & modify label
604                             sprintf(newLabel, "!!!-> %s", itemLabel);

605                             ReplaceListItem (pnlSetProgram,
                            pnlProgram_lstProgSteps, j, newLabel,
                            stepRef[j][0]);
606                         }
607
608                         stepRef[j][1] = status;
609                     }
610                 }
```

```
611             }
612         }
613
614
615         if(errFlag) MessagePopup ("State Missing", "A state that was
            used by this program has been deleted!\n          Please select a
            new state for the affected step" );
616
617         return 0;
618     }
619
620
621     int CVICALLBACK SetupNewButtons (int panel, int control, int event,
622             void *callbackData, int eventData1, int eventData2)
623     {
624         int numButtons, bSize;
625
626         if(event != EVENT_COMMIT) return 0;
627
628
629         GetCtrlVal(pnlButtonSetup, btnPanel_numSolenoids, &numSolenoids
            );
630         GetCtrlVal (pnlButtonSetup, btnPanel_numButtonSize, &bSize);
631
632         DestroyArray(0);
633         CreateButtonArray(numSolenoids, bSize);
634         activeControl = 0;
635
636
637         RemovePopup (pnlButtonSetup);
638
639         return 0;
640     }
641
642     int CVICALLBACK changeButtonSize (int panel, int control, int event,
643             void *callbackData, int eventData1, int eventData2)
644     {
645         int scaleFactor = 2;
646         int newScale, left, top, dX, dY;
647
648         if(event != EVENT_COMMIT) return 0;
649
650         GetCtrlVal (pnlButtonSetup, btnPanel_numButtonSize, &newScale);
651         GetCtrlAttribute (pnlButtonSetup, btnPanel_DECORATION, ATTR_TOP
            , &top);
652         GetCtrlAttribute (pnlButtonSetup, btnPanel_DECORATION,
            ATTR_LEFT, &left);
```

```
653         GetCtrlAttribute (pnlButtonSetup, btnPanel_DECORATION,
            ATTR_HEIGHT, &dY);
654         GetCtrlAttribute (pnlButtonSetup, btnPanel_DECORATION,
            ATTR_WIDTH, &dX);
655
656         SetCtrlAttribute (pnlButtonSetup, btnPanel_sampleLED,
            ATTR_HEIGHT, (scaleFactor*solenoidSize[--newScale]));
657         SetCtrlAttribute (pnlButtonSetup, btnPanel_sampleLED,
            ATTR_WIDTH, solenoidSize[newScale]);
658 659
660         SetCtrlAttribute (pnlButtonSetup, btnPanel_sampleLED, ATTR_LEFT
            , (left + (int)(0.5*dX) - (int)(0.5*solenoidSize[newScale])));
661         SetCtrlAttribute (pnlButtonSetup, btnPanel_sampleLED, ATTR_TOP,
             (top + (int)(0.5*dY) - (int)(2*0.5*solenoidSize[newScale])));
662
663         return 0;
664     }
665
666     int CVICALLBACK loadImage (int panel, int control, int event,
667             void *callbackData, int eventData1, int eventData2)
668     {
669         int status = -1;
670
671         if(event != EVENT_COMMIT) return 0;
672
673
674         status = FileSelectPopup ("", "*.bmp", "*.bmp", "Select Image",
             VAL_LOAD_BUTTON, 0, 1, 1, 0, imageFileName);
675         if(status == 0) return 0;
676
677         DisplayImageFile (mainPanel, mainPanel_pictScheme,
            imageFileName);
678
679         return 0;
680     }
681
682
683     int findDuplicateState(char* usrName)
684     {
685         int i, numStates;
686         char lstStateName[100];
687
688         GetNumListItems (pnlSetStates, pnlStates_lstStates, &numStates
            );                  //get total number of states
689         for(i = 0; i < numStates; i++)
                                                                    //for each
```

```
        entry
690     {   GetLabelFromIndex (pnlSetStates, pnlStates_lstStates, i,
        lstStateName);       // get name
691         if(strcmp(lstStateName, usrName) == 0) return 1;
                                          // compare to usrName
692     }

                        //     return 1 if duplicate
693
694     return 0;
695 }

                    //else return 0
696
697
698 void getNewStateName(char* newName)
699 {
700     int i = 2;
701
702
703     sprintf(newName, "new state");
704     while(findDuplicateState(newName))
705         sprintf(newName, "new state%i", i++);
706
707     return;
708 }
709
710 int getUniqueID()
711 {
712     int i, numStates, flag, newID;
713     srand(time(NULL));
714
715     GetCtrlVal(pnlSetStates, pnlStates_numStates, &numStates);
716     flag = 1;
717     while(flag)
718     {   flag = 0;
719         newID = rand() % 1000;
720         for(i = 0; i < numStates; i++)
721             if(newID == uID[i]) flag = 1;
722     }
723
724     return newID;
725 }
726
727
728 int CVICALLBACK changeNumStates (int panel, int control, int event,
729         void *callbackData, int eventData1, int eventData2)
```

```
730 {
731     int numStates, existingStates;
732     int i, j, k, m;
733     char newStateName[100];
734
735     if(event != EVENT_COMMIT) return 0;
736
737
738     GetCtrlVal(pnlSetStates, pnlStates_numStates, &numStates);
739     GetNumListItems(pnlSetStates, pnlStates_lstStates, &
        existingStates);
740
741     j = 0; m = 0;
742     for(i = 0; i < (numStates - existingStates); i++)
                                          //when states need to be added
743     {   while(solenoidState[j][0] != -1) j++;
                                                  // find open space in
        solenoid state array
744         stateList[existingStates+i] = solenoidState[j];
                                            // and assign to stateList
745         solenoidState[j][0] = 0;
                                                                      //
            indicate occupied
746         getNewStateName(newStateName);
                                                                 // get new
            name
747         uID[existingStates+i] = getUniqueID();
                                                       // and unique ID
748         InsertListItem (pnlSetStates, pnlStates_lstStates, (
            existingStates + i), newStateName, (existingStates + i));
749         if(existingStates > 0)
750             for(k = 0; k < numSolenoids; k++) solenoidState[j][k] =
                 stateList[existingStates+i-1][k];       // initialize to
                previous step's configuration
751     }
752
753     for(i = 0; i < (existingStates - numStates); i++)
                                             //when states need to be
        removed
754     {   stateList[existingStates - i - 1][0] = -1;
                                               //open up space in
        solenoidState[][]
755         stateList[existingStates - i - 1] = NULL;
                                                 //remove reference
756         uID[existingStates - i - 1] = -1;
                                                                  //remove uID
757         DeleteListItem (pnlSetStates, pnlStates_lstStates, (
```

```
            existingStates - 1), 1);        //remove from list
758     }
759
760     return 0;
761 }
762
763 int CVICALLBACK ChooseState (int panel, int control, int event,
764         void *callbackData, int eventData1, int eventData2)
765 {
766     int i, index, oldValue;
767     int color[3];
768     char stepName[20];
769
770
771
772     if(event == EVENT_LEFT_DOUBLE_CLICK)
773     {   GetCtrlIndex(pnlSetStates, pnlStates_lstStates, &index);
                            //get active index
774         GetValueFromIndex (pnlSetStates, pnlStates_lstStates, index
            , &oldValue);
775         PromptPopup ("Step Name", "Enter a new label for this step"
            , stepName, sizeof(stepName) - 1);
776         while(findDuplicateState(stepName))
777             PromptPopup ("Step Name", "Enter a unique label for
                this step", stepName, sizeof(stepName) - 1);
778         ReplaceListItem (pnlSetStates, pnlStates_lstStates, index,
            stepName, oldValue);            //rename
779         return 0;
780     }
781
782
783     if(event != EVENT_VAL_CHANGED) return 0;
784
785     color[0] = VAL_TRANSPARENT;
786     color[1] = activeColor;

        //initialize color array
787     color[2] = ncColor;
788
789     GetCtrlIndex(pnlSetStates, pnlStates_lstStates, &index);
                                //get active index
790     currState = stateList[index];
                                                                //set
        "current state" pointer to new index
791
792     for(i = 0; i < numSolenoids; i++)
793     {   SetCtrlVal(mainPanel, solenoids[i][0], currState[i]);
```

```
794             SetCtrlAttribute (mainPanel, solenoids[i][3],
                ATTR_FRAME_COLOR, color[currState[i]]);
795             SetCtrlAttribute (mainPanel, solenoids[i][4],
                ATTR_TEXT_BGCOLOR, color[currState[i]]);
796             SetCtrlAttribute (mainPanel, solenoids[i][0], ATTR_ON_COLOR
                , color[currState[i]]);
797         }
798
799         return 0;
800     }
801
802     int CVICALLBACK removeState (int panel, int control, int event,
803             void *callbackData, int eventData1, int eventData2)
804     {
805         int i, index, numSteps;
806
807         if(event != EVENT_COMMIT) return 0;
808
809         GetNumListItems(pnlSetStates, pnlStates_lstStates, &numSteps);
                    //get total steps
810         GetCtrlIndex(pnlSetStates, pnlStates_lstStates, &index);
                          //get active index
811         currState = stateList[index];
                                                          //set "current
        state" pointer to new index
812
813         currState[0] = -1;
                                                                  //open
        up space in solenoidState[][]
814         currState = NULL;
                                                                   //and
        remove ptr
815
816         SetCtrlVal(pnlSetStates, pnlStates_numStates, --numSteps);
                        //update front panel
817         DeleteListItem (pnlSetStates, pnlStates_lstStates, index, 1);
                     //erase from list
818         for(i = index; i < numSteps; i++)
819         {    stateList[i] = stateList[i+1];
                                                      //reassign pointers
820             uID[i] = uID[i+1];
                                                                  //& uIDs
821         }
822
823
824         return 0;
825     }
```

```
826
827  int CVICALLBACK insertState (int panel, int control, int event,
828           void *callbackData, int eventData1, int eventData2)
829  {
830       int i, j, k, index, numSteps;
831       char stateName[100];
832
833       if(event != EVENT_COMMIT) return 0;
834
835
836       GetNumListItems(pnlSetStates, pnlStates_lstStates, &numSteps);
                  //get total steps
837       GetCtrlIndex(pnlSetStates, pnlStates_lstStates, &index);
                        //get active index
838       SetCtrlVal(pnlSetStates, pnlStates_numStates, ++numSteps);
                      //update front panel
839       getNewStateName(stateName);
                                                                  //get unique
       state name
840       index++;

       //insert item after current index instead of before
841       InsertListItem(pnlSetStates, pnlStates_lstStates, index,
       stateName, index);
842       SetCtrlIndex(pnlSetStates, pnlStates_lstStates, index);
843
844       for(i = numSteps-1; i > index; i--)
845       {    stateList[i] = stateList[i-1];
                                                 // reassign pointers for
       rest of array
846            uID[i] = uID[i-1];
847       }
848
849       j = 0;
850       while(solenoidState[j][0] != -1) j++;
                                              // find open space in
       solenoid state array
851       stateList[index] = solenoidState[j];
                                               // and assign to stateList
852       uID[index] = getUniqueID();
853       if(index > 0)
854            for(k = 0; k < numSolenoids; k++) solenoidState[j][k] =
            stateList[index-1][k];          // initialize to previous
            step's configuration
855       else solenoidState[j][0] = 0;


856
```

```
857
858
859         return 0;
860     }
861
862     int CVICALLBACK moveStateUp (int panel, int control, int event,
863             void *callbackData, int eventData1, int eventData2)
864     {
865         int i, index, swapID;
866         int* swapPtr;
867         char label[100], prevLabel[100];
868
869         if(event != EVENT_COMMIT) return 0;
870
871         GetCtrlIndex(pnlSetStates, pnlStates_lstStates, &index);
                            //get active index
872         if(index == 0) return 0;
                                                                    //if already
            at top, exit
873
874         GetLabelFromIndex (pnlSetStates, pnlStates_lstStates, index,
            label);      //record old values
875         GetLabelFromIndex (pnlSetStates, pnlStates_lstStates, index-1,
            prevLabel);
876         swapPtr = stateList[index];
877         swapID = uID[index];
878
879         ReplaceListItem (pnlSetStates, pnlStates_lstStates, index,
            prevLabel, index);
880         stateList[index] = stateList[index-1];
881         uID[index] = uID[index-1];
882
883         ReplaceListItem (pnlSetStates, pnlStates_lstStates, index-1,
            label, index-1);
884         stateList[index-1] = swapPtr;
885         uID[index-1] = swapID;
886
887         SetCtrlIndex (pnlSetStates, pnlStates_lstStates, index-1);
                          //simulate click on swapped val
888         ChooseState(pnlSetStates, pnlStates_lstStates,
            EVENT_VAL_CHANGED, NULL, 0, 0);
889
890         return 0;
891     }
892
893     int CVICALLBACK moveStateDown (int panel, int control, int event,
894             void *callbackData, int eventData1, int eventData2)
```

```
895  {
896      int i, index, numSteps, swapID;
897      int* swapPtr;
898      char label[100], nextLabel[100];
899
900      if(event != EVENT_COMMIT) return 0;
901
902      GetNumListItems(pnlSetStates, pnlStates_lstStates, &numSteps);
                  //get total steps
903      GetCtrlIndex(pnlSetStates, pnlStates_lstStates, &index);
                        //get active index
904      if(index == numSteps-1) return 0;
                                                  //if already at top, exit
905
906      GetLabelFromIndex (pnlSetStates, pnlStates_lstStates, index,
         label);      //record old values
907      GetLabelFromIndex (pnlSetStates, pnlStates_lstStates, index+1,
         nextLabel);
908      swapPtr = stateList[index];
909      swapID = uID[index];
910
911      ReplaceListItem (pnlSetStates, pnlStates_lstStates, index,
         nextLabel, index);
912      stateList[index] = stateList[index+1];
913      uID[index] = uID[index+1];
914
915      ReplaceListItem (pnlSetStates, pnlStates_lstStates, index+1,
         label, index+1);
916      stateList[index+1] = swapPtr;
917      uID[index+1] = swapID;
918
919      SetCtrlIndex (pnlSetStates, pnlStates_lstStates, index+1);
                     //simulate click on swapped val
920      ChooseState(pnlSetStates, pnlStates_lstStates,
         EVENT_VAL_CHANGED, NULL, 0, 0);
921
922      return 0;
923  }
924
925
926  int CVICALLBACK loadSetup (int panel, int control, int event,
927           void *callbackData, int eventData1, int eventData2)
928  {
929      int i;
930
931      if(event != EVENT_COMMIT) return 0;
932
```

```
933
934
935         SetCtrlAttribute (mainPanel, mainPanel_cmdSetup, ATTR_DIMMED, 1
            );
936         SetCtrlAttribute (mainPanel, mainPanel_cmdDefineStates,
            ATTR_DIMMED, 0);
937         SetCtrlAttribute (mainPanel, mainPanel_cmdDefineProgram,
            ATTR_DIMMED, 0);
938
939
940         HidePanel (pnlSetStates);
941         HidePanel (pnlSetProgram);
942         activeColor = VAL_RED;
943         clickMode = 0;
944
945         for(i = 0; i < numSolenoids; i++)
                                                    // turn off "activated"
            solenoids
946         {    SetCtrlVal (mainPanel, solenoids[i][0], 0);

947              SetCtrlAttribute (mainPanel, solenoids[i][3],
                 ATTR_FRAME_COLOR, VAL_TRANSPARENT);
948              SetCtrlAttribute (mainPanel, solenoids[i][4],
                 ATTR_TEXT_BGCOLOR, VAL_TRANSPARENT);
949         }
950
951         i = 0;
952         while((i < numSolenoids) && (solenoids[i][0] != -1))
953         {    SetCtrlAttribute (mainPanel, solenoids[i][0],
            ATTR_LABEL_VISIBLE, 0);
954              SetCtrlAttribute (mainPanel, solenoids[i++][0],
                 ATTR_ON_COLOR, activeColor);
955         }
956
957         return 0;
958    }
959
960    int CVICALLBACK loadStates (int panel, int control, int event,
961              void *callbackData, int eventData1, int eventData2)
962    {
963         int i, index;
964
965         if(event != EVENT_COMMIT) return 0;
966
967
968         SetCtrlAttribute (mainPanel, mainPanel_cmdSetup, ATTR_DIMMED, 0
            );
```

```
969         SetCtrlAttribute (mainPanel, mainPanel_cmdDefineStates,
            ATTR_DIMMED, 1);
970         SetCtrlAttribute (mainPanel, mainPanel_cmdDefineProgram,
            ATTR_DIMMED, 0);
971
972
973         DisplayPanel (pnlSetStates);
974         HidePanel (pnlSetProgram);
975         activeColor = VAL_GREEN;
976         clickMode = 1;
977
978         if(activeControl)
979         {    SetCtrlVal (mainPanel, activeControl, 0);
                                        // turn off "active" button
980              index = activeSolenoid();
981              SetCtrlAttribute (mainPanel, solenoids[index][3],
                 ATTR_FRAME_COLOR, VAL_TRANSPARENT);
982              SetCtrlAttribute (mainPanel, solenoids[index][4],
                 ATTR_TEXT_BGCOLOR, VAL_TRANSPARENT);
983         }
984
985         changeNumStates(pnlSetStates, pnlStates_numStates, EVENT_COMMIT
            , NULL, 0, 0);      // generate first state if not present
986         SetCtrlIndex(pnlSetStates, pnlStates_lstStates, 0);
                                          // auto-select first state
987         ChooseState(pnlSetStates, pnlStates_lstStates,
            EVENT_VAL_CHANGED, NULL, 0, 0); // simulate click
988
989
990         i = 0;
991         while((i < numSolenoids) && (solenoids[i][0] != -1))
992         {    SetCtrlAttribute (mainPanel, solenoids[i][0],
            ATTR_LABEL_BGCOLOR, VAL_TRANSPARENT);
993              SetCtrlAttribute (mainPanel, solenoids[i][0],
                 ATTR_LABEL_VISIBLE, 0);
994              SetCtrlAttribute (mainPanel, solenoids[i++][0],
                 ATTR_ON_COLOR, activeColor);
995         }
996
997         return 0;
998    }
999
1000   int CVICALLBACK loadProgram (int panel, int control, int event,
1001             void *callbackData, int eventData1, int eventData2)
1002   {
1003        int i, index;
1004
```

```
1005        if(event != EVENT_COMMIT) return 0;
1006
1007
1008        SetCtrlAttribute (mainPanel, mainPanel_cmdSetup, ATTR_DIMMED, 0
            );
1009        SetCtrlAttribute (mainPanel, mainPanel_cmdDefineStates,
            ATTR_DIMMED, 0);
1010        SetCtrlAttribute (mainPanel, mainPanel_cmdDefineProgram,
            ATTR_DIMMED, 1);
1011
1012        if(activeControl)
1013        {   SetCtrlVal (mainPanel, activeControl, 0);
                                        // turn off "active" button
1014            index = activeSolenoid();
1015            SetCtrlAttribute (mainPanel, solenoids[index][3],
                ATTR_FRAME_COLOR, VAL_TRANSPARENT);
1016            SetCtrlAttribute (mainPanel, solenoids[index][4],
                ATTR_TEXT_BGCOLOR, VAL_TRANSPARENT);
1017        }
1018
1019        buildStatesRing();
1020        refreshStepLinks();
1021
1022        DisplayPanel (pnlSetProgram);
1023        HidePanel (pnlSetStates);
1024        activeColor = MakeColor (255, 128, 0);
1025        clickMode = 2;
1026        goStatus = -1;
1027
1028        i = 0;
1029        while((i < numSolenoids) && (solenoids[i][0] != -1))
                                //go through solenoids
1030        {   SetCtrlVal (mainPanel, solenoids[i][0], 0);
                                    //turn each one "off"
1031            SetCtrlAttribute (mainPanel, solenoids[i][3],
                ATTR_FRAME_COLOR, VAL_TRANSPARENT);
1032            SetCtrlAttribute (mainPanel, solenoids[i][4],
                ATTR_TEXT_BGCOLOR, VAL_TRANSPARENT);
1033            SetCtrlAttribute (mainPanel, solenoids[i][0],
                ATTR_LABEL_VISIBLE, 1);               //enable lighted border
1034            SetCtrlAttribute (mainPanel, solenoids[i++][0],
                ATTR_ON_COLOR, activeColor);       //and set new active color
1035        }
1036
1037        return 0;
1038    }
1039
```

```
1040  int saveStateFile(char* filePath)
1041  {
1042      int i, j, status, bSize;
1043      int varX, varY, flag;
1044      char tmpString[50], picName[MAX_FILENAME_LEN];
1045      FILE* saveFile;
1046
1047
1048      saveFile = fopen (filePath, "w");
1049
              //open file for write
1050      SplitPath (imageFileName, NULL, NULL, picName);
1051      fprintf(saveFile, "imgFile: %s\n", picName);
1052      GetCtrlVal (pnlButtonSetup, btnPanel_numButtonSize, &bSize);
                   //get button size
1053      fprintf(saveFile, "solenoids: %i %i\n", numSolenoids, bSize);
                  //write number & size to file
1054
1055      for(i = 0; i < numSolenoids; i++)
                                                    //write button locations
1056      {    GetCtrlAttribute (mainPanel, solenoids[i][0], ATTR_HEIGHT,
          &varX);
1057           GetCtrlAttribute (mainPanel, solenoids[i][0], ATTR_WIDTH, &
               varY);
1058           flag = (varX < varY ? 0 : 1);
                                                        // note if button
               is rotated
1059           fprintf(saveFile, "%i %i %i\n", solenoids[i][1], solenoids[
               i][2], flag);
1060      }
1061
1062
1063      GetNumListItems(pnlSetStates, pnlStates_lstStates, &bSize);
1064      fprintf(saveFile, "numStates: %i\n", bSize);
1065      for(i = 0; i < bSize; i++)
                                                      //write states
1066      {    GetLabelFromIndex(pnlSetStates, pnlStates_lstStates, i,
          tmpString);
1067           fprintf(saveFile, "%s: (%03i); ", tmpString, uID[i]);

1068           for(j = 0; j < numSolenoids; j++)
1069               fprintf(saveFile, "%i ", stateList[i][j]);
1070           fprintf(saveFile, "\n");
1071      }
1072
1073      fflush(saveFile);
1074      fclose(saveFile);
```

```
1075
1076        return 0;
1077    }
1078
1079 1080
1081    int loadStateFile(char* filePath)
1082    {
1083        int i, j, k, status, bSize;
1084        int varX, varY, flag;
1085        char readString[500];
1086        char stateName[30], junk[500];
1087        char volume[MAX_DRIVENAME_LEN], fileDir[MAX_DIRNAME_LEN],
          picName[MAX_FILENAME_LEN];
1088        char* token;
1089        FILE* dataFile;
1090
1091
1092
1093        dataFile = fopen (filePath, "r");
1094
1095        fgets (readString, sizeof(readString), dataFile);
1096            SplitPath (filePath, volume, fileDir, NULL);
                                        //get current directory
1097            for(i = 9; i < strlen(readString); i++)
                                              //get
            filename

1098                picName[i-9] = readString[i];
1099                picName[i-10] = '\0';


1100            if(strlen(picName) != 0)
1101            {   sprintf(imageFileName, "%s%s%s", volume, fileDir,
            picName);       //construct picture path
1102                DisplayImageFile (mainPanel, mainPanel_pictScheme,
                imageFileName);
1103            }
1104
1105        fgets (readString, sizeof(readString), dataFile);
1106            sscanf(readString, "solenoids: %i %i\n", &numSolenoids, &
            bSize);
1107            DestroyArray(1);
1108            CreateButtonArray(numSolenoids, bSize);
1109            SetCtrlVal (pnlButtonSetup, btnPanel_numButtonSize, bSize);
1110        for(i = 0; i < numSolenoids; i++)
                                                    //
```

```
        right click)
1111    {   fgets (readString, sizeof(readString), dataFile);
1112        sscanf(readString, "%i %i %i", &solenoids[i][1], &solenoids
            [i][2], &flag);
1113        if(solenoids[i][1] != -1) SetCtrlAttribute(mainPanel,
            solenoids[i][0], ATTR_TOP, solenoids[i][2]);
1114        if(solenoids[i][2] != -1) SetCtrlAttribute(mainPanel,
            solenoids[i][0], ATTR_LEFT, solenoids[i][1]);
1115        if(flag)
1116        {   GetCtrlAttribute (mainPanel, solenoids[i][0],
            ATTR_HEIGHT, &varY);
1117            GetCtrlAttribute (mainPanel, solenoids[i][0],
                ATTR_WIDTH, &varX);
1118
1119            SetCtrlAttribute (mainPanel, solenoids[i][0],
                ATTR_HEIGHT, varX);
1120            SetCtrlAttribute (mainPanel, solenoids[i][0],
                ATTR_WIDTH, varY);
1121            SetCtrlAttribute (mainPanel, solenoids[i][0],
                ATTR_LABEL_WIDTH, varY);
1122        }
1123        SetCtrlAttribute (mainPanel, solenoids[i][0], ATTR_ON_COLOR
            , VAL_GREEN);
1124    }
1125
1126 1127
1128    fgets (readString, sizeof(readString), dataFile);
1129        sscanf(readString, "numStates: %i\n", &bSize);
1130        ClearListCtrl(pnlSetStates, pnlStates_lstStates);
1131    for(i = 0; i < bSize; i++)
1132    {   fgets (readString,sizeof(readString), dataFile);
1133        j = 0;
1134        while(readString[j] != ':') stateName[j] = readString[j++];
1135        stateName[j] = '\0';
1136        InsertListItem (pnlSetStates, pnlStates_lstStates, i,
            stateName, i);
1137        stateList[i] = solenoidState[i];
1138        k = j = j + 3;
1139        while(readString[k] != ')') junk[k-j] = readString[k++];
1140        junk[k] = '\0';
1141        uID[i] = atoi(junk);
1142        token = strtok (readString, ";");
1143        token = strtok (NULL, " ");
                                                    //first token is
            state name & ID
1144        j = 0;
```

```
1145            while(token != NULL)
1146            {   sscanf(token, "%i", &stateList[i][j++]);
1147                token = strtok (NULL, " ");
1148            }
1149        }
1150        SetCtrlVal (pnlSetStates, pnlStates_numStates, bSize);
1151
1152        fclose (dataFile);
1153
1154        return 0;
1155    }
1156
1157
1158    int CVICALLBACK saveStates (int panel, int control, int event,
1159            void *callbackData, int eventData1, int eventData2)
1160    {
1161        int status;
1162        char filePath[MAX_PATHNAME_LEN];
1163
1164
1165        if(event != EVENT_COMMIT) return 0;
1166
1167        status = FileSelectPopup ("", "*.sta", "*.sta", "Save states
          file", VAL_SAVE_BUTTON, 0, 1, 1, 1, filePath);
1168        if(status < 1) return 0;
1169
1170        saveStateFile(filePath);
1171
1172        return 0;
1173    }
1174
1175    int CVICALLBACK loadStatesfromFile (int panel, int control, int
    event,
1176            void *callbackData, int eventData1, int eventData2)
1177    {
1178        int status;
1179        long fileSize;
1180        char filePath[MAX_PATHNAME_LEN];
1181
1182        if(event != EVENT_COMMIT) return 0;
1183
1184
1185
1186        status = FileSelectPopup ("", "*.sta", "*.sta", "Load states
          file", VAL_LOAD_BUTTON, 0, 1, 1, 0, filePath);
1187        if(status < 1) return 0;
1188        if(!(GetFileInfo (filePath, &fileSize)))
```

```
                                                    //if file doesn't exist
1189        { MessagePopup ("Error", "There was a problem opening the
            file!");
1190            return 0;
1191        }
1192
1193        loadStateFile(filePath);
1194
1195        return 0;
1196    }
1197
1198    int CVICALLBACK addNewStep (int panel, int control, int event,
1199             void *callbackData, int eventData1, int eventData2)
1200    {
1201        int newIndex;
1202
1203        if(event != EVENT_COMMIT) return 0;
1204
1205
1206
1207        GetNumListItems (pnlSetProgram, pnlProgram_lstProgSteps, &
            newIndex);
1208        SetCtrlVal (pnlEditStep, pnlEdtStep_numEditIndex, newIndex);
1209        SetCtrlVal (pnlEditStep, pnlEdtStep_chkMode, 0);
1210        SetCtrlAttribute (pnlEditStep, pnlEdtStep_cmdEditStep,
            ATTR_LABEL_TEXT, "Add Step");
1211
1212        InstallPopup (pnlEditStep);
1213
1214        return 0;
1215    }
1216
1217    int CVICALLBACK cancelStepEdit (int panel, int control, int event,
1218             void *callbackData, int eventData1, int eventData2)
1219    {
1220        if(event != EVENT_COMMIT) return 0;
1221
1222             RemovePopup (pnlEditStep);
1223
1224        return 0;
1225    }
1226
1227    int CVICALLBACK editStep (int panel, int control, int event,
1228             void *callbackData, int eventData1, int eventData2)
1229    {
1230        char stepLabel[100], stepName[30], durString[10];
1231        int stateIndex, duration, usrIndex, editMode;
```

```
1232        int i, temp;
1233
1234
1235        if(event != EVENT_COMMIT) return 0;
1236
1237        GetCtrlVal    (pnlEditStep, pnlEdtStep_chkMode, &editMode);
1238        GetCtrlVal    (pnlEditStep, pnlEdtStep_numEditIndex, &usrIndex);
1239        GetCtrlVal    (pnlEditStep, pnlEdtStep_numStateDuration, &duration
            );
1240        GetCtrlVal    (pnlEditStep, pnlEdtStep_rngStates, &stateIndex);
1241
1242
1243
1244
1245        if(stateIndex == -1)
1246        {
1247            if(editMode) ReplaceListItem (pnlSetProgram,
                pnlProgram_lstProgSteps, usrIndex, "Programmed pause",
                PAUSECODE);
1248            else
1249            {    InsertListItem (pnlSetProgram, pnlProgram_lstProgSteps,
                 usrIndex, "Programmed pause", PAUSECODE);
1250                CheckListItem (pnlSetProgram, pnlProgram_lstProgSteps,
                    usrIndex, 1);
1251            }
1252            stepRef[usrIndex][0] = stepRef[usrIndex][1] = stepRef[
                usrIndex][2] = PAUSECODE;
1253        }
1254        else
1255        {    GetLabelFromIndex (pnlSetStates, pnlStates_lstStates,
            stateIndex, stepName);
1256            strcpy(stepLabel, stepName);
1257            temp = strlen(stepName);
1258            for(i = temp; i < 22; i++)
1259                stepLabel[i] = ' ';
1260            stepLabel[i] = '\0';
1261            sprintf(durString, "%i", duration);
1262            strcat(stepLabel, durString);
1263
1264            if(editMode) ReplaceListItem (pnlSetProgram,
                pnlProgram_lstProgSteps, usrIndex, stepLabel, uID[
                stateIndex]);
1265            else
1266            {    InsertListItem (pnlSetProgram, pnlProgram_lstProgSteps,
                 usrIndex, stepLabel, uID[stateIndex]);
1267                CheckListItem (pnlSetProgram, pnlProgram_lstProgSteps,
                    usrIndex, 1);
```

```
1268            }
1269            stepRef[usrIndex][0] = uID[stateIndex];
1270            stepRef[usrIndex][1] = stateIndex;
1271            stepRef[usrIndex][2] = duration;
1272        }
1273
1274
1275        RemovePopup (0);
1276
1277        return 0;
1278    }
1279
1280 1281
1282    int CVICALLBACK dimDuration (int panel, int control, int event,
1283            void *callbackData, int eventData1, int eventData2)
1284    {
1285        int stateIndex;
1286
1287        if(event != EVENT_COMMIT) return 0;
1288
1289        GetCtrlVal (pnlEditStep, pnlEdtStep_rngStates, &stateIndex);
1290        if(stateIndex == -1)
1291        {   SetCtrlAttribute (pnlEditStep, pnlEdtStep_numStateDuration,
             ATTR_DIMMED, 1);
1292            SetCtrlAttribute (pnlEditStep, pnlEdtStep_txtDuration,
                ATTR_DIMMED, 1);
1293        }
1294        else
1295        {   SetCtrlAttribute (pnlEditStep, pnlEdtStep_numStateDuration,
             ATTR_DIMMED, 0);
1296            SetCtrlAttribute (pnlEditStep, pnlEdtStep_txtDuration,
                ATTR_DIMMED, 0);
1297        }
1298
1299        return 0;
1300    }
1301
1302
1303    int CVICALLBACK ChooseStep (int panel, int control, int event,
1304            void *callbackData, int eventData1, int eventData2)
1305    {
1306        int i, index, stateIndex;
1307        int color[3];
1308        char stepName[20];
1309
1310
```

```
1311
1312
1313        if(event == EVENT_LEFT_DOUBLE_CLICK)
1314        {    if(goStatus == 2) return 0;
                                                                    //if run
            active, return
1315
1316            GetCtrlIndex(pnlSetProgram, pnlProgram_lstProgSteps, &index
                );                  //get active index
1317            SetCtrlIndex(pnlEditStep, pnlEdtStep_rngStates, (stepRef[
                index][1] + 1));      //initialize values
1318            SetCtrlVal (pnlEditStep, pnlEdtStep_numStateDuration,
                stepRef[index][2]);
1319            SetCtrlVal (pnlEditStep, pnlEdtStep_numEditIndex, index);
1320            SetCtrlVal (pnlEditStep, pnlEdtStep_chkMode, 1);
1321            SetCtrlAttribute (pnlEditStep, pnlEdtStep_cmdEditStep,
                ATTR_LABEL_TEXT, "Edit Step");
1322
1323            InstallPopup (pnlEditStep);
1324            return 0;
1325        }
1326
1327
1328        if(event != EVENT_VAL_CHANGED) return 0;
1329
1330        color[0] = VAL_TRANSPARENT;
1331        color[1] = activeColor;
                                                                    //initialize
            color array
1332        color[2] = ncColor;
1333
1334        GetCtrlIndex(pnlSetProgram, pnlProgram_lstProgSteps, &index);
                    //get active index
1335        if(index == -1) return 0;
1336        if(stepRef[index][1] == PAUSECODE)
1337        {    SetCtrlAttribute (mainPanel, mainPanel_txtUserIntervention ,
             ATTR_VISIBLE, 1);
1338            return 0;
1339        }
1340        else SetCtrlAttribute (mainPanel, mainPanel_txtUserIntervention
            , ATTR_VISIBLE, 0);
1341        currState = stateList[stepRef[index][1]];
                                            //set "current state" pointer
            to new index
1342
1343        for(i = 0; i < numSolenoids; i++)
1344        {    if(eventData1 == 1)
```

```
            //if run executed
1345        {   if(currState[i] != 2) SetCtrlVal(mainPanel, solenoids[i
            ][0], currState[i]);        // if state is not "maintain
            prior", update screen
1346        }
1347        else SetCtrlAttribute (mainPanel, solenoids[i][0],
            ATTR_LABEL_BGCOLOR, color[currState[i]]);
1348        SetCtrlAttribute (mainPanel, solenoids[i][3],
            ATTR_FRAME_COLOR, color[currState[i]]);
1349        SetCtrlAttribute (mainPanel, solenoids[i][4],
            ATTR_TEXT_BGCOLOR, color[currState[i]]);
1350    }
1351
1352    return 0;
1353}
1354
1355int CVICALLBACK removeStep (int panel, int control, int event,
1356         void *callbackData, int eventData1, int eventData2)
1357{
1358    char stepLabel[100], stepName[30], durString[10];
1359    int currIndex, numSteps;
1360    int i, temp;
1361
1362
1363    if(event != EVENT_COMMIT) return 0;
1364
1365    GetCtrlIndex (pnlSetProgram, pnlProgram_lstProgSteps, &
    currIndex);
1366    GetNumListItems (pnlSetProgram, pnlProgram_lstProgSteps, &
    numSteps);
1367
1368    for(i = currIndex; i < (numSteps-1); i++)
                                        //shift references to swallow
    deleted
1369    {   stepRef[i][0] = stepRef[i+1][0]; 1370
    stepRef[i][1] = stepRef[i+1][1]; 1371
    stepRef[i][2] = stepRef[i+1][2];
1372    }
1373    stepRef[i][0] = stepRef[i][1] = stepRef[i][2] = -1;
                            //and remove last reference
1374
1375    DeleteListItem (pnlSetProgram, pnlProgram_lstProgSteps,
    currIndex, 1); //remove list item
1376    ChooseStep (panel, control, EVENT_VAL_CHANGED, NULL, 0, 0);
1377
1378    return 0;
```

```
1379    }
1380
1381    int CVICALLBACK moveStepUp (int panel, int control, int event,
1382             void *callbackData, int eventData1, int eventData2)
1383    {
1384        int i, index;
1385        int swapStore[3];
1386        char label[100], prevLabel[100];
1387
1388        if(event != EVENT_COMMIT) return 0;
1389
1390        GetCtrlIndex(pnlSetProgram, pnlProgram_lstProgSteps, &index);
                    //get active index
1391        if(index == 0) return 0;
                                                                //if already
          at top, exit
1392
1393        GetLabelFromIndex (pnlSetProgram, pnlProgram_lstProgSteps,
          index, label);          //record old values
1394        GetLabelFromIndex (pnlSetProgram, pnlProgram_lstProgSteps,
          index-1, prevLabel);
1395        for(i = 0; i < 3; i++) swapStore[i] = stepRef[index][i];
1396
1397
1398        ReplaceListItem (pnlSetProgram, pnlProgram_lstProgSteps, index,
           prevLabel, index);
1399        for(i = 0; i < 3; i++) stepRef[index][i] = stepRef[index-1][i];
1400
1401        ReplaceListItem (pnlSetProgram, pnlProgram_lstProgSteps, index-
          1, label, index-1);
1402        for(i = 0; i < 3; i++) stepRef[index-1][i] = swapStore[i];
1403
1404        SetCtrlIndex (pnlSetProgram, pnlProgram_lstProgSteps, index-1);
                    //simulate click on swapped val
1405        ChooseStep(pnlSetProgram, pnlProgram_lstProgSteps,
          EVENT_VAL_CHANGED, NULL, 0, 0);
1406 1407
1408        return 0;
1409    }
1410
1411    int CVICALLBACK moveStepDown (int panel, int control, int event,
1412             void *callbackData, int eventData1, int eventData2)
1413    {
1414        int i, index, numItems;
1415        int swapStore[3];
1416        char label[100], nextLabel[100];
```

```
1417
1418        if(event != EVENT_COMMIT) return 0;
1419
1420        GetCtrlIndex(pnlSetProgram, pnlProgram_lstProgSteps, &index);
                    //get active index
1421        GetNumListItems(pnlSetProgram, pnlProgram_lstProgSteps, &
            numItems);
1422        if(index == (numItems-1)) return 0;
                                                  //if already at bottom,
            exit
1423
1424        GetLabelFromIndex (pnlSetProgram, pnlProgram_lstProgSteps,
            index, label);          //record old values
1425        GetLabelFromIndex (pnlSetProgram, pnlProgram_lstProgSteps,
            index+1, nextLabel);
1426        for(i = 0; i < 3; i++) swapStore[i] = stepRef[index][i];
1427
1428
1429        ReplaceListItem (pnlSetProgram, pnlProgram_lstProgSteps, index,
             nextLabel, index);
1430        for(i = 0; i < 3; i++) stepRef[index][i] = stepRef[index+1][i];
1431
1432        ReplaceListItem (pnlSetProgram, pnlProgram_lstProgSteps, index+
            1, label, index+1);
1433        for(i = 0; i < 3; i++) stepRef[index+1][i] = swapStore[i];
1434
1435        SetCtrlIndex (pnlSetProgram, pnlProgram_lstProgSteps, index+1);
                    //simulate click on swapped val
1436        ChooseStep(pnlSetProgram, pnlProgram_lstProgSteps,
            EVENT_VAL_CHANGED, NULL, 0, 0);
1437 1438
1439        return 0;
1440    }
1441
1442 1443
1444    int CVICALLBACK startRun (int panel, int control, int event,
1445             void *callbackData, int eventData1, int eventData2)
1446    {
1447        int i;
1448        int currStep, totalSteps, checked;
1449        double startTime, currTime, stepTime;
1450
1451        if(event != EVENT_COMMIT) return 0;
1452
1453
```

```
1454
1455        SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdPlay,
            ATTR_DIMMED, 1);          //prevent second run start
1456        SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdPause,
            ATTR_DIMMED, 0);         //enable run-control buttons
1457        SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdStop,
            ATTR_DIMMED, 0);
1458        SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdForceNext,
            ATTR_DIMMED, 0);
1459        SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdForcePrev,
            ATTR_DIMMED, 0);
1460        SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdRemoveStep,
            ATTR_DIMMED, 1);
1461        SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdAddStep,
            ATTR_DIMMED, 1);
1462        SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdMoveStepUp,
            ATTR_DIMMED, 1);
1463        SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdMoveStepDown,
            ATTR_DIMMED, 1);
1464        SetCtrlAttribute (mainPanel, mainPanel_cmdDefineStates,
            ATTR_DIMMED, 1);
1465        SetCtrlAttribute (mainPanel, mainPanel_cmdSetup, ATTR_DIMMED, 1
            );
1466        SetCtrlAttribute (pnlSetProgram, pnlProgram_lstProgSteps,
            ATTR_CTRL_MODE, VAL_INDICATOR);       //no changes allowed during
            run
1467        SetCtrlAttribute (mainPanel, mainPanel_progressBar,
            ATTR_VISIBLE, 1);
1468
1469        GetNumListItems (pnlSetProgram, pnlProgram_lstProgSteps, &
            totalSteps);
1470        currStep = 0;
1471        stepOverride = 0;
1472        goStatus = 2;
1473
1474        while(currStep < totalSteps)
1475        {    IsListItemChecked (pnlSetProgram, pnlProgram_lstProgSteps,
            currStep, &checked);                 //only process checked steps
1476            if(checked)
1477            {    SetCtrlIndex (pnlSetProgram, pnlProgram_lstProgSteps,
                currStep);                              // select state
1478                ChooseStep (pnlSetProgram, pnlProgram_lstProgSteps,
                EVENT_VAL_CHANGED, NULL, 1, 0);         // & update on
                screen
1479                ProcessDrawEvents ();
1480                if(stepRef[currStep][1] == PAUSECODE)
                                                                        //
```

```
                    if pause
1481                {    pauseRun(0, 0, EVENT_COMMIT, NULL, 0, 0);
                                                                    //
                    "click" button
1482                     stepRef[currStep][2] = 1;
1483                }
1484                else
1485                {    setState(stepRef[currStep][1]);         }
                                          // set solenoids' config
1486
1487
1488                stepTime = stepRef[currStep][2];
1489                startTime = currTime = Timer ();
1490                while((currTime - startTime) < stepTime)
                                           // while waiting for step delay
1491                {    ProcessSystemEvents();
                                                       //     get user
                    events
1492                     switch(goStatus)
                                                                    //
                     check status
1493                     {    case 2:

                     //       if normal
1494                              currTime = Timer();
                                                                     //
                              record newest time
1495                              SetCtrlVal (mainPanel,
                              mainPanel_progressBar, 100*(currTime-
                              startTime)/stepTime);
1496                              break;
1497                         case 0:

                              //       if pause
1498                              stepTime -= (currTime - startTime);
                                                  //          record time
                              already taken
1499                              currTime = startTime;
                                                                    //
                              kill clock (time taken = 0)
1500                              break;
1501                         case 1:

                              //       if resume
1502                              startTime = currTime = Timer();
                                                        //          resync
                              clock to curr time
```

```
1503                            goStatus = 2;

                                //          resume normal state
1504                            break;
1505                        case -1:

                            //       if stop
1506                            MessagePopup ("Run Complete", "Your run
                                has been aborted!");        //          give msg
1507                            stepOverride = totalSteps;
                                                                  //
                                overload steps to quit loop
1508                            break;
1509                    }
1510
1511                    if(stepOverride) stepTime = 0;
                                                          //    if FF/RW end
                        curr step
1512                }
1513            }
1514
1515            if(stepOverride)
                                                                        //   if
            FF/RW ended step
1516            {    currStep += stepOverride;
                                                           //     make adjustment
1517                stepOverride = 0;
                                                                      //
                clear flag
1518            }
1519            else currStep++;
                                                                        //
            else move to next step
1520        }
1521
1522
1523        SetCtrlAttribute    (pnlSetProgram, pnlProgram_cmdPlay,
            ATTR_DIMMED, 0);      //re-enable start run
1524        SetCtrlAttribute    (pnlSetProgram, pnlProgram_cmdPause,
            ATTR_DIMMED, 1);     //disable run-control buttons
1525        SetCtrlAttribute    (pnlSetProgram, pnlProgram_cmdStop,
            ATTR_DIMMED, 1);
1526        SetCtrlAttribute    (pnlSetProgram, pnlProgram_cmdForceNext,
            ATTR_DIMMED, 1);
1527        SetCtrlAttribute    (pnlSetProgram, pnlProgram_cmdForcePrev,
            ATTR_DIMMED, 1);
1528        SetCtrlAttribute    (pnlSetProgram, pnlProgram_cmdRemoveStep,
```

```
        ATTR_DIMMED, 0);
1529    SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdAddStep,
        ATTR_DIMMED, 0);
1530    SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdMoveStepUp,
        ATTR_DIMMED, 0);
1531    SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdMoveStepDown,
        ATTR_DIMMED, 0);
1532    SetCtrlAttribute (mainPanel, mainPanel_cmdDefineStates,
        ATTR_DIMMED, 0);
1533    SetCtrlAttribute (mainPanel, mainPanel_cmdSetup, ATTR_DIMMED, 0
        );
1534    SetCtrlAttribute (pnlSetProgram, pnlProgram_lstProgSteps,
        ATTR_CTRL_MODE, VAL_HOT);
1535    SetCtrlAttribute (mainPanel, mainPanel_progressBar,
        ATTR_VISIBLE, 0);
1536 1537
1538    if(goStatus != -1) MessagePopup ("Run Complete", "Your run has
        completed successfully!");
1539    goStatus = -1;
1540
1541    return 0;
1542  }
1543
1544  int CVICALLBACK pauseRun (int panel, int control, int event,
1545          void *callbackData, int eventData1, int eventData2)
1546  {
1547    if(event != EVENT_COMMIT) return 0;
1548
1549    if(goStatus == 2)

                        //if normal run in progress
1550    {   goStatus = 0;

                        // set pause flag
1551        SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdPause,
            ATTR_IMAGE_FILE, pauseActive);            // swap button pic
1552        SetCtrlAttribute (pnlSetProgram, pnlProgram_lstProgSteps,
            ATTR_CTRL_MODE, VAL_HOT);             // allow check/uncheck
            action
1553    }
1554    else

                                    //else run resume
1555    {   goStatus = 1;

                        //   set resume flag
```

```
1556            SetCtrlAttribute (pnlSetProgram, pnlProgram_cmdPause,
                ATTR_IMAGE_FILE, pauseNormal);            // swap button pic
1557            SetCtrlAttribute (pnlSetProgram, pnlProgram_lstProgSteps,
                ATTR_CTRL_MODE, VAL_INDICATOR);       // no changes
                allowed!
1558        }
1559
1560        return 0;
1561    }
1562
1563    int CVICALLBACK stopRun (int panel, int control, int event,
1564            void *callbackData, int eventData1, int eventData2)
1565    {
1566        if(event != EVENT_COMMIT) return 0;
1567
1568        goStatus = -1;
1569
1570        return 0;
1571    }
1572
1573    int CVICALLBACK setManualCtrl (int panel, int control, int event,
1574            void *callbackData, int eventData1, int eventData2)
1575    {
1576        int state;
1577
1578        if(event != EVENT_COMMIT) return 0;
1579
1580        GetCtrlVal(pnlSetProgram, pnlProgram_chkEnableManualCtrl , &
                state);
1581        SetCtrlAttribute(pnlSetProgram, pnlProgram_chkModifySolenoids ,
                ATTR_DIMMED, !state);
1582        SetCtrlAttribute(pnlSetProgram, pnlProgram_lstLoadState,
                ATTR_DIMMED, !state);
1583        clickMode = 2;
1584
1585        return 0;
1586    }
1587
1588    int CVICALLBACK setSolenoidPolicy (int panel, int control, int
    event,
1589            void *callbackData, int eventData1, int eventData2)
1590    {
1591        int value;
1592
1593        if(event != EVENT_COMMIT) return 0;
1594
1595
```

```
1596        GetCtrlVal(pnlSetProgram, pnlProgram_chkModifySolenoids , &value
            );
1597        clickMode = (value > 0)?3:2;
1598
1599        return 0;
1600    }
1601
1602    int CVICALLBACK loadState (int panel, int control, int event,
1603            void *callbackData, int eventData1, int eventData2)
1604    {
1605        int state, index, i;
1606        int color[3];                   //color[2]
1607
1608        if(event != EVENT_COMMIT) return 0;
1609
1610        GetCtrlVal(pnlSetProgram, pnlProgram_lstLoadState, &state);
1611        setState(state);
1612
1613        color[0] = VAL_TRANSPARENT;
1614        color[1] = activeColor;
                                                                //initialize
            color array
1615        color[2] = ncColor;
1616
1617        if(state == -1) return 0;
1618        currState = stateList[state];
                                                        //set "current state"
            pointer to new index
1619
1620        for(i = 0; i < numSolenoids; i++)
1621        {    if(currState[i] != 2) SetCtrlVal(mainPanel, solenoids[i][0
            ], currState[i]);
1622            SetCtrlAttribute (mainPanel, solenoids[i][3],
                ATTR_FRAME_COLOR, color[currState[i]]);
1623            SetCtrlAttribute (mainPanel, solenoids[i][4],
                ATTR_TEXT_BGCOLOR, color[currState[i]]);
1624        }
1625        return 0;
1626    }
1627
1628    int CVICALLBACK JumpStep (int panel, int control, int event,
1629            void *callbackData, int eventData1, int eventData2)
1630    {
1631        if(event != EVENT_COMMIT) return 0;
1632
1633        stepOverride = 1;
                                                                //indicate
```

```
        fwd to next step
1634
1635        return 0;
1636    }
1637
1638    int CVICALLBACK repeatStep (int panel, int control, int event,
1639            void *callbackData, int eventData1, int eventData2)
1640    {
1641        if(event != EVENT_COMMIT) return 0;
1642
1643        stepOverride = -1;
                                                                    //indicate
        return to prev step
1644
1645        return 0;
1646    }
1647
1648    int CVICALLBACK saveProgram (int panel, int control, int event,
1649            void *callbackData, int eventData1, int eventData2)
1650    {
1651        int i, status, bSize;
1652        char filePath[MAX_PATHNAME_LEN], stateFilePath[MAX_PATHNAME_LEN
        ], tmpString[50];
1653        FILE* saveFile, stateFile;
1654
1655        if(event != EVENT_COMMIT) return 0;
1656
1657        status = FileSelectPopup ("", "*.prg", "*.prg", "Save program
        file", VAL_SAVE_BUTTON, 0, 1, 1, 1, filePath);
1658        if(status < 1) return 0;
1659
1660        strcpy(stateFilePath, filePath);
1661        stateFilePath[strlen(stateFilePath) - 4] = '\0';
                              //remove extension
1662        strcat(stateFilePath, ".sta");
                                                  //add state file extension
1663        saveStateFile(stateFilePath);
                                                   //and save state file
        parameters
1664
1665
1666        saveFile = fopen (filePath, "w");
1667
        //open file for write
1668
1669    //  fprintf(saveFile, "State File: %s\n", stateFilePath);
1670
```

```
1671        GetNumListItems(pnlSetProgram, pnlProgram_lstProgSteps, &bSize);
1672        fprintf(saveFile, "numSteps: %i\n", bSize);
1673        for(i = 0; i < bSize; i++)
                                                        //write states
1674        {   GetLabelFromIndex(pnlSetProgram, pnlProgram_lstProgSteps, i
            , tmpString);
1675            IsListItemChecked(pnlSetProgram, pnlProgram_lstProgSteps, i
                , &status);
1676
1677            fprintf(saveFile, "%s: %i %i %i %i\n", tmpString, status,
                stepRef[i][0], stepRef[i][1], stepRef[i][2]);
1678        }
1679
1680        fflush(saveFile); 1681
            fclose(saveFile);
1682
1683
1684        return 0;
1685    }
1686
1687    int CVICALLBACK loadProgramfromFile (int panel, int control, int
    event,
1688             void *callbackData, int eventData1, int eventData2)
1689    {
1690        int i, j, k, status, bSize;
1691        long fileSize;
1692        char filePath[MAX_PATHNAME_LEN], stateFilePath[MAX_PATHNAME_LEN
            ], readString[300], labelStr[50];
1693        char driveName[MAX_DRIVENAME_LEN], dirName[MAX_DIRNAME_LEN],
            fileName[MAX_FILENAME_LEN];
1694        FILE* dataFile;
1695
1696        if(event != EVENT_COMMIT) return 0;
1697
1698
1699        status = FileSelectPopup ("", "*.prg", "*.prg", "Load program
            file", VAL_LOAD_BUTTON, 0, 1, 1, 0, filePath);
1700        if(status < 1) return 0;
1701        if(!(GetFileInfo (filePath, &fileSize)))
                                        //if file doesn't exist
1702        { MessagePopup ("Error", "There was a problem opening the
            program file!");
1703            return 0;
1704        }
1705
1706        dataFile = fopen (filePath, "r");
1707
```

```
1708
1709        strncpy(stateFilePath, filePath, (strlen(filePath) - 3));
                    //copy into state file name, truncate extension
1710        stateFilePath[strlen(filePath)-3] = '\0';
1711        strcat(stateFilePath, "sta");
                                                    //add ".sta"
            extension
1712        loadStateFile(stateFilePath);
1713
1714        buildStatesRing();
1715        refreshStepLinks();
1716
1717        ClearListCtrl (pnlSetProgram, pnlProgram_lstProgSteps);
1718
1719        fgets (readString, sizeof(readString), dataFile);
                              //read in first line
1720        sscanf (readString, "numSteps: %i", &bSize);
                                    //extract number of steps
1721
1722        for(i = 0; i < bSize; i++)
                                                          //for each step
1723        {   fgets (readString, sizeof(readString), dataFile);
                         // grab line
1724            j = 0;
1725            while(readString[j] != ':') labelStr[j] = readString[j++];
                   // find colon
1726            labelStr[j] = '\0';

1727            k = j;

                // record position
1728            while(readString[j] != '\0') readString[j-k] = readString[j
                ++]; // cut out all text prior
1729            readString[j-k] = '\0';
1730
1731            sscanf(readString, ": %i %i %i %i", &status, &stepRef[i][0
                ], &stepRef[i][1], &stepRef[i][2]);
1732
1733            InsertListItem (pnlSetProgram, pnlProgram_lstProgSteps, i,
                labelStr, stepRef[i][0]);
1734            CheckListItem (pnlSetProgram, pnlProgram_lstProgSteps, i,
                status);
1735        }
1736
1737        i = 0;
1738        activeColor = MakeColor (255, 128, 0);
1739        while((i < numSolenoids) && (solenoids[i][0] != -1))
```

```
                        //go through solenoids
1740        {   SetCtrlVal (mainPanel, solenoids[i][0], 0);
                            //turn each one "off"
1741            SetCtrlAttribute (mainPanel, solenoids[i][3],
                ATTR_FRAME_COLOR, VAL_TRANSPARENT);
1742            SetCtrlAttribute (mainPanel, solenoids[i][4],
                ATTR_TEXT_BGCOLOR, VAL_TRANSPARENT);
1743            SetCtrlAttribute (mainPanel, solenoids[i][0],
                ATTR_LABEL_VISIBLE, 1);
1744            SetCtrlAttribute (mainPanel, solenoids[i++][0],
                ATTR_ON_COLOR, activeColor);        //and set new active color
1745        }
1746
1747        return 0;
1748    }
1749
1750    int CVICALLBACK chkSize (int panel, int control, int event,
1751             void *callbackData, int eventData1, int eventData2)
1752    {
1753        int newVal;
1754
1755        if(event != EVENT_COMMIT) return 0;
1756
1757        GetCtrlVal(panel, control, &newVal);
1758        if(newVal < 24) SetCtrlAttribute (panel, btnPanel_numButtonSize
            , ATTR_MAX_VALUE, 5);
1759        else if(newVal < 26) SetCtrlAttribute (panel,
            btnPanel_numButtonSize, ATTR_MAX_VALUE, 4);
1760        else if(newVal < 29) SetCtrlAttribute (panel,
            btnPanel_numButtonSize, ATTR_MAX_VALUE, 3);
1761        else SetCtrlAttribute (panel, btnPanel_numButtonSize,
            ATTR_MAX_VALUE, 2);      //if(newVal < 33) SetCtrlAttribute
            (panel, btnPanel_numButtonSize, ATTR_MAX_VALUE, 2);
1762
1763        return 0;
1764    }
1765
```

```
#include   <rs232.h>
#include   <formatio.h>
#include   <ansi_c.h>
#include   <utility.h>
#include   <cvirte.h>
#include   <userint.h>
#include   "mainPanel.h"
#include   "mapSolenoids.h"

#define   MAXCHIPS 3
#define   MAXSOLENOIDS 64
#define   MAXSTATES 100
#define   MAXSTEPS 100
#define   PAUSECODE -5


static int pnlMain;
static int pnlMapping;


int loadStateFile (char*);
void setupMapping(void);
int setup(void);


int usbPort;

int solenoids[MAXSOLENOIDS];
int SCsolenoidConfig[MAXSOLENOIDS][4];
                                                    //ctrl ID, Top, Left,
orientation
int solenoidLUT[MAXCHIPS][MAXSOLENOIDS];

int solenoidState[MAXSTATES][MAXSOLENOIDS];
                                  //solenoid state for each program step
int uID[MAXSTATES];
                                                              //unique ID
for each state
int* stateList[MAXSTATES];
                                                     //correlate list to
solenoidState[]
int stepRef[MAXSTEPS][3];
                                                      //[0] state uID,
[1] reference to stateList [2] duration for each step
char stepNames[MAXSTEPS][50];
                                                  //name associated with
each step
```

```
37  int solenoidConfig[MAXSOLENOIDS];
                                                    //current configuration of
    all solenoids. Need this information for "Ignore" solenoids
38  int instanceIndex[MAXCHIPS] = {0, 1, 2};
                                               //instance of chip, used as
    callback data for timers, buttons
39  int ctrlArrays[MAXCHIPS][7];
                                                          //[0] border, [1] step
    bar, [2] overall bar, [3] start button, [4] timer, [5] status, [6]
    steps list
40  int currStep[MAXCHIPS];
                                                             //keep track of
    current step for each instance
41  int runActive[MAXCHIPS];
                                                           //should mainTimer
    a chip's update progress bars?
42
43  int activeSCsolenoid = -1;
                                                            //active single-chip
    solenoid
44  int activeABSsolenoid = -1;
                                                           //active absolute
    solenoid
45
46
47
48  /****** DLP232 commands ******/
49  int inhibitOn = 112, inhibitOff = 113, pulseVon = 116, pulseVoff =
    38;
50  int busLines[8][2] = {81, 49, 87, 50, 69, 51, 82, 52, 84, 53, 89,
    54, 85, 55, 73, 56};          //array[bus line][off/on]
51  int ctrlLines[3][2] = {105, 104, 101, 100, 97, 47};
52  int chipID[8][3] = {0,0,0, 0,0,1, 0,1,0, 0,1,1, 1,0,0, 1,0,1, 1,1,0
    , 1,1,1};
53  /*******************************/
54
55
56
57
58
59
60
61  int main (int argc, char *argv[])
62  {
63      if (InitCVIRTE (0, argv, 0) == 0)
64          return -1; /* out of memory */
65      if ((pnlMain = LoadPanel (0, "mainPanel.uir", mainPanel)) < 0)
```

```
66              return -1;
67          if ((pnlMapping = LoadPanel (0, "mapSolenoids.uir", mapPanel))
            < 0)
68              return -1;
69
70
71          setup();
72
73          DisplayPanel (pnlMain);
74          RunUserInterface (); 75
            DiscardPanel (pnlMain);
76          return 0;
77      }
78
79 80 81
82      int loadState(int stateNum, int chipNum)
83      {
84          int i, j, numBanks, offset;
85
86          for(i = 0; i < 8; i++)

                //for each bank
87          {   offset = 8*i;
88              for(j = 0; j < 3; j++) ComWrtByte (usbPort, ctrlLines[j][
                chipID[i][j]]);               // select chip
89              for(j = 0; j < 8; j++) ComWrtByte (usbPort, busLines[j][
                solenoidConfig[offset+j]]); // load all current values to
                bus
90              for(j = 0; j < MAXSOLENOIDS; j++)
91              {   if((solenoidLUT[chipNum][j] > offset-1) && (solenoidLUT
                [chipNum][j] < offset + 8) && (solenoidState[stateNum][j]
                != 2)) //if solenoid falls into currently-selected chip
                range
92                  {   ComWrtByte (usbPort, busLines[solenoidLUT[chipNum][
                    j] % 8][solenoidState[stateNum][j]]);
                    //     else write new value
93                      solenoidConfig[solenoidLUT[chipNum][j]] =
                        solenoidState[stateNum][j];                    //
                        and record to memory
94                  }
95              }
96
97
98              ComWrtByte (usbPort, inhibitOff);
                                                                        //
```

```
            latch values
 99         Delay(0.05);
            //Delay(0.005);

100         ComWrtByte (usbPort, inhibitOn);
101     }
102
103     ComWrtByte (usbPort, pulseVon);

        //switch values
104     Delay(0.05);              //Delay(0.001);
105     ComWrtByte (usbPort, pulseVoff);
106
107
108     return 0;
109 }
110
111 112
113 int allSolenoidsOff(int chipNum)
114 {
115     int i, j, numBanks, offset;
116
117     for(i = 0; i < 8; i++)

            //for each bank
118     {   offset = 8*i;
119         for(j = 0; j < 3; j++) ComWrtByte (usbPort, ctrlLines[j][
            chipID[i][j]]);              // select chip
120         for(j = 0; j < 8; j++) ComWrtByte (usbPort, busLines[j][
            solenoidConfig[offset+j]]); // load all current values to
            bus
121         for(j = 0; j < MAXSOLENOIDS; j++)
122         {   if((solenoidLUT[chipNum][j] > offset-1) && (solenoidLUT
            [chipNum][j] < offset + 8)) //if solenoid falls into
            currently-selected chip range
123             {   ComWrtByte (usbPort, busLines[solenoidLUT[chipNum][
                j] % 8][0]);               //    else write new value
124                 solenoidConfig[solenoidLUT[chipNum][j]] = 0;
                                        //    and record to memory
125             }
126         }
127
128
129         ComWrtByte (usbPort, inhibitOff);
                                                                    //
            latch values
```

```
130             Delay(0.05);
                //Delay(0.005);

131             ComWrtByte (usbPort, inhibitOn);
132         }
133
134         ComWrtByte (usbPort, pulseVon);

            //switch values
135         Delay(0.05);                //Delay(0.001);
136         ComWrtByte (usbPort, pulseVoff);
137
138         return 0;
139     }
140
141 142 143
144     void initSolenoids()
145     {
146         int i, j, numBanks, offset;
147
148         for(i = 0; i < 8; i++)

                //for each bank
149         {   offset = 8*i;
150             for(j = 0; j < 3; j++) ComWrtByte (usbPort, ctrlLines[j][
                chipID[i][j]]);                 // select chip
151             for(j = 0; j < 8; j++) ComWrtByte (usbPort, busLines[j][0
                ]);                             // set all solenoids low
152
153
154             ComWrtByte (usbPort, inhibitOff);
                                                                        //
                latch values
155             Delay(0.05);
                //Delay(0.005);

156             ComWrtByte (usbPort, inhibitOn);
157         }
158
159         ComWrtByte (usbPort, pulseVon);

            //switch values
160         Delay(0.05);                //Delay(0.001);
161         ComWrtByte (usbPort, pulseVoff);
162
```

```
163         return;
164     }
165
166 167 168
169     int openUSB()
170     {
171         int i, success, response;
172         int comPort = -1;
173         int ping = 39;

                        //ascii code for apostrophe (')
174
175
176         for(i = 3; i < 4; i++)
177     // for(i = 0; i < 10;
        i++)
            //open com ports sequentially
178         {    success = OpenComConfig (i, "", 460800, 0, 8, 1, 512, 512);
                                        // open port
179             if(success == 0)

                        // if successfull
180             {    while(ComWrtByte (i, ping) != 1);
                                                                    //     issue
                ping
181                 response = ComRdByte (i);

                    //     read response
182                 if(response == 'Q')

                            //     if ping response from DLP232
183                 {    comPort = i;

                            //       record port number
184                     i = 10;

                                                //       exit loop
185                 }
186                 CloseCom (i);

                                //     else close com port
187             }
188         }
189
190         return comPort;
```

```
191  }
192
193
194
195
196  int setup()
197  {
198       int i,j;
199
200
201       SetCtrlAttribute (pnlMain, mainPanel_cmdStartChip1,
         ATTR_CALLBACK_DATA, &instanceIndex[0]);                  //register
         controls for each chip
202       SetCtrlAttribute (pnlMain, mainPanel_cmdStartChip2,
         ATTR_CALLBACK_DATA, &instanceIndex[1]);                  //so that
         shared callback can differentiate
203       SetCtrlAttribute (pnlMain, mainPanel_cmdStartChip3,
         ATTR_CALLBACK_DATA, &instanceIndex[2]);
204
205       SetCtrlAttribute (pnlMain, mainPanel_Timer1, ATTR_CALLBACK_DATA
         , &instanceIndex[0]);
206       SetCtrlAttribute (pnlMain, mainPanel_Timer2, ATTR_CALLBACK_DATA
         , &instanceIndex[1]);
207       SetCtrlAttribute (pnlMain, mainPanel_Timer3, ATTR_CALLBACK_DATA
         , &instanceIndex[2]);
208
209
210
211
212       ctrlArrays[0][0] = mainPanel_border1;
                                                                  //set up
         control arrays
213       ctrlArrays[1][0]    = mainPanel_border2; 214
         ctrlArrays[2][0]    = mainPanel_border3;
215       ctrlArrays[0][1]    = mainPanel_progressBar11;
216       ctrlArrays[1][1]    = mainPanel_progressBar21;
217       ctrlArrays[2][1]    = mainPanel_progressBar31;
218       ctrlArrays[0][2]    = mainPanel_progressBar12;
219       ctrlArrays[1][2]    = mainPanel_progressBar22;
220       ctrlArrays[2][2]    = mainPanel_progressBar32;
221       ctrlArrays[0][3]    = mainPanel_cmdStartChip1;
222       ctrlArrays[1][3]    = mainPanel_cmdStartChip2;
223       ctrlArrays[2][3]    = mainPanel_cmdStartChip3;
224       ctrlArrays[0][4]    = mainPanel_Timer1;
225       ctrlArrays[1][4]    = mainPanel_Timer2;
226       ctrlArrays[2][4]    = mainPanel_Timer3;
227       ctrlArrays[0][5]    = mainPanel_chipStatus1;
```

```
228         ctrlArrays[1][5]    = mainPanel_chipStatus2;
229         ctrlArrays[2][5]    = mainPanel_chipStatus3;
230         ctrlArrays[0][6]    = mainPanel_rngSteps1;
231         ctrlArrays[1][6]    = mainPanel_rngSteps2;
232         ctrlArrays[2][6]    = mainPanel_rngSteps3;
233
234
235         for(i = 0; i < MAXCHIPS; i++)
236         {    SetCtrlAttribute (pnlMain, ctrlArrays[i][0],
            ATTR_FRAME_COLOR, VAL_TRANSPARENT);
237              runActive[i] = 0;
238              currStep[i] = 0;
239         }
240
241
242
243
                                 //initialize data arrays
244
245         for(i = 0; i < MAXSTATES; i++)
246         {    stepRef[i][0] = -1;
247              solenoidState[i][0] = -1;

                 //-1 indicates open slot
248              for(j=0;j<MAXSOLENOIDS;j++)

                 //unnecessary, but just initialize everything here
249                  solenoidState[i][j] = 0;
250              stateList[i] = NULL;
251              uID[i] = -1;
252         }
253
254 255
256    //   setStatus("Finding USB device");
257         ProcessDrawEvents();
258         usbPort = openUSB();

            //find DLP232PC
259         if(usbPort == -1)
260         {    MessagePopup("Error", "The hardware device could not be
            located!");                 //if not found, give msg
261              return 1;
262         }
263         else

                                 //otherwise
```

```
264         {    SetCtrlAttribute (pnlMain, mainPanel_cmdLoadRunFile,
            ATTR_DIMMED, 0);              // enable run controls
265             ComWrtByte (usbPort, 117);

                //'u' = no analog channels
266             ComWrtByte (usbPort, inhibitOn);

267         }
268
269         initSolenoids();
270     //  setStatus("");
271
272
273         return 0;
274     }
275
276
277
278
279     int CVICALLBACK quitProgram (int panel, int control, int event,
280             void *callbackData, int eventData1, int eventData2)
281     {
282         if(event != EVENT_COMMIT) return 0;
283
284         QuitUserInterface(0);
285
286         return 0;
287     }
288
289     int CVICALLBACK MapButtons (int panel, int control, int event,
290             void *callbackData, int eventData1, int eventData2)
291     {
292         if(event != EVENT_COMMIT) return 0;
293
294         InstallPopup (pnlMapping);
295         setupMapping();
296     //  loadStateFile();
297
298
299         return 0;
300     }
301
302
303
304     int loadStatesFromFile(char* filePath)
305     {
306
```

```
307         int i, j, k, status, bSize;
308         int varX, varY, flag;
309         int numSolenoids;
310         char readString[500];
311         char stateName[30], junk[500];
312         char volume[MAX_DRIVENAME_LEN], fileDir[MAX_DIRNAME_LEN],
            picName[MAX_FILENAME_LEN];
313         char* token;
314         FILE* dataFile;
315
316
317
318
319         dataFile = fopen (filePath, "r");
320
321         fgets (readString, sizeof(readString), dataFile);
                                              //read in picture name
322         fgets (readString, sizeof(readString), dataFile);
                                              //read in number of solenoids
323             sscanf(readString, "solenoids: %i %i\n", &numSolenoids, &
                bSize);
324         for(i = 0; i < numSolenoids; i++)

325             fgets (readString, sizeof(readString), dataFile);
                                              //read in solenoid positioning
                data
326
327
328         fgets (readString, sizeof(readString), dataFile);
                                              //here's where the relevant
            data starts
329             sscanf(readString, "numStates: %i\n", &bSize);
                                                 //number of states
330         for(i = 0; i < bSize; i++)
331         {   fgets (readString,sizeof(readString), dataFile);
332             j = 0;
333             while(readString[j] != ':') stateName[j] = readString[j++];
334             stateName[j] = '\0';
335             stateList[i] = solenoidState[i];
336             k = j = j + 3;
337             while(readString[k] != ')') junk[k-j] = readString[k++];
338             junk[k] = '\0';
339             uID[i] = atoi(junk);
340             token = strtok (readString, ";");
341             token = strtok (NULL, " ");
                                                                    //first
                token is state name & ID
```

```
342             j = 0;
343             while(token != NULL)
344             {    sscanf(token, "%i", &stateList[i][j++]);

345                  token = strtok (NULL, " ");
346             }
347         }
348
349         fclose (dataFile);
350
351         return 0;
352     }
353
354
355     int loadMappingFromFile(char* filePath)
356     {
357
358         int i, j, k, status, bSize;
359         char readString[500];
360         char stateName[30], junk[500];
361         char* token;
362         FILE* dataFile;
363
364
365
366         dataFile = fopen (filePath, "r");
367
                        //I will assume    no. chips/solenoids do not
368
                        // change, so I read in but don't record
369
                        // values.
370         fgets (readString, sizeof(readString), dataFile);
                                              //read in MAXCHIPS
371         fgets (readString, sizeof(readString), dataFile);
                                              //read in MAXSOLENOIDS
372

373         for(i = 0; i < MAXCHIPS; i++)

374         {    fgets (readString, sizeof(readString), dataFile);
                                         //read in solenoid positioning data
375             token = strtok (readString, " ");
376             j = 0;
377             while(token != NULL)
378             {    sscanf(token, "%i", &solenoidLUT[i][j++]);
379                  token = strtok (NULL, " ");
```

```
380             }
381         }
382
383
384         fclose (dataFile);
385
386         return 0;
387     }
388
389 390
391     int CVICALLBACK Load_Runfile (int panel, int control, int event,
392             void *callbackData, int eventData1, int eventData2)
393     {
394         int i, j, k, status, bSize;
395         long fileSize;
396         char filePath[MAX_PATHNAME_LEN], stateFilePath[MAX_PATHNAME_LEN ],
            mappingFilePath[MAX_PATHNAME_LEN], readString[300], labelStr
            [50];
397         char driveName[MAX_DRIVENAME_LEN], dirName[MAX_DIRNAME_LEN],
            fileName[MAX_FILENAME_LEN];
398         FILE* dataFile;
399
400         if(event != EVENT_COMMIT) return 0;
401
402
403         status = FileSelectPopup ("", "*.prg", "*.prg", "Load program
            file", VAL_LOAD_BUTTON, 0, 1, 1, 0, filePath);
404         if(status < 1) return 0;
405         if(!(GetFileInfo (filePath, &fileSize)))
                                                    //if file doesn't exist
406         { MessagePopup ("Error", "There was a problem opening the
            program file!");
407             return 0;
408         }
409
410         dataFile = fopen (filePath, "r");
411
412
413         strncpy(stateFilePath, filePath, (strlen(filePath) - 3));
                                    //copy into state file name, truncate
            extension
414         stateFilePath[strlen(filePath)-3] = '\0';
415         strcat(stateFilePath, "sta");
                                                                  //add
            ".sta" extension
416         loadStatesFromFile(stateFilePath);
```

```
417
418         strncpy(stateFilePath, filePath, (strlen(filePath) - 3));
                                    //copy into mapping file name, truncate
        extension
419         stateFilePath[strlen(filePath)-3] = '\0';
420         strcat(stateFilePath, "smf");
                                                                    //add
        ".solenoid mapping file" extension
421         loadMappingFromFile(stateFilePath);
422
423
424
425         fgets (readString, sizeof(readString), dataFile);
                                              //read in first line
426         sscanf (readString, "numSteps: %i", &bSize);
                                                  //extract number of steps
427
428         for(i = 0; i < bSize; i++)
                                                                        //for
        each step
429         {    fgets (readString, sizeof(readString), dataFile);
                                         // grab line
430             j = 0;
431             while(readString[j] != ':') labelStr[j] = readString[j++];
                                        // find colon
432             labelStr[j] = '\0';
433             strcpy(stepNames[i], labelStr);
434             k = j;

                                    // record position
435             while(readString[j] != '\0') readString[j-k] = readString[j
                ++];                // cut out all text prior
436             readString[j-k] = '\0';
437
438             sscanf(readString, ": %i %i %i %i", &status, &stepRef[i][0
                ], &stepRef[i][1], &stepRef[i][2]);
439
440         }
441
442 443
444         for(i = 0; i < bSize; i++)
                                                                        //ensure
        that runfile links to state list are accurate
445             if((stepRef[i][0] > -1) && (stepRef[i][0] != uID[stepRef[i
                ][1]])) i = bSize + 5;           // if error, quit loop
446         if(i == bSize + 5)
```

```
447         {    MessagePopup ("Error!", "Error building runfile - please
            regenerate your source files!" );
448              return 1;
449         }
450
451
452         for(i = 0; i < 3; i++)

            //enable start buttons
453              SetCtrlAttribute(mainPanel, ctrlArrays[i][3], ATTR_DIMMED,
                 0);
454
455         return 0;
456    }
457
458 459 460
461    int CVICALLBACK releaseChip (int panelHandle, int controlID, int
       event, void *callbackData, int eventData1, int eventData2)
462    {
463         int* chipNo;
464         int chip;
465
466         if(event != EVENT_COMMIT) return 0;
467
468         chipNo = callbackData;
469         chip = *chipNo;
470
471
472         SetCtrlAttribute (mainPanel, ctrlArrays[chip][3],
            ATTR_LABEL_TEXT, "Start Run");
473         InstallCtrlCallback (mainPanel, ctrlArrays[chip][3], startRun,
            &instanceIndex[chip]);
474         SetCtrlAttribute (mainPanel, ctrlArrays[chip][3], ATTR_DIMMED,
            0);
475         SetCtrlAttribute (mainPanel, ctrlArrays[chip][0],
            ATTR_FRAME_COLOR, VAL_TRANSPARENT);
476         SetCtrlAttribute (mainPanel, ctrlArrays[chip][1], ATTR_DIMMED,
            1);
477         SetCtrlAttribute (mainPanel, ctrlArrays[chip][2], ATTR_DIMMED,
            1);
478         SetCtrlAttribute (mainPanel, ctrlArrays[chip][6], ATTR_DIMMED,
            1);
479         SetCtrlVal (mainPanel, ctrlArrays[chip][1], 0);
480         SetCtrlVal (mainPanel, ctrlArrays[chip][2], 0);
481         ClearListCtrl (mainPanel, ctrlArrays[chip][6]);
```

```
482         runActive[chip] = 0;
483         allSolenoidsOff(chip);
484 
485         ProcessSystemEvents ();
486 
487         return 0;
488     }
489 
490 
491     int CVICALLBACK startRun (int panel, int control, int event,
492             void *callbackData, int eventData1, int eventData2)
493     {
494         int* chipNo;
495         int chip;
496         int numSteps = -1,
497             totalTime = 0;
498 
499         if(event != EVENT_COMMIT) return 0;
500 
501         chipNo = callbackData;
                                                                    //can't
        dereference void* directly, so copy to int*
502         chip = *chipNo;
503 
504 
505 
506 
507         while(stepRef[++numSteps][0] != -1)
                                                    //get number of steps
        & total run time
508             if(stepRef[numSteps][2] > 0) totalTime += stepRef[numSteps
                ][2];
509 
510         SetCtrlAttribute (mainPanel, ctrlArrays[chip][2],
        ATTR_MAX_VALUE, totalTime);           //set up progress bars
511         SetCtrlAttribute (mainPanel, ctrlArrays[chip][1],
        ATTR_MAX_VALUE, stepRef[0][2]);
512         SetCtrlVal(mainPanel, ctrlArrays[chip][2], 0);
513         SetCtrlVal(mainPanel, ctrlArrays[chip][1], 0);
514         SetCtrlAttribute (mainPanel, ctrlArrays[chip][4], ATTR_INTERVAL
        , (double)stepRef[0][2]);       //set up timer
515         SetCtrlAttribute (mainPanel, ctrlArrays[chip][3], ATTR_DIMMED,
        1);           //disable button
516         SetCtrlAttribute (mainPanel, ctrlArrays[chip][1], ATTR_DIMMED,
        0);
517         SetCtrlAttribute (mainPanel, ctrlArrays[chip][2], ATTR_DIMMED,
        0);
```

```
518         SetCtrlAttribute (mainPanel, ctrlArrays[chip][6], ATTR_DIMMED,
            0);
519         SetCtrlAttribute (mainPanel, ctrlArrays[chip][0],
            ATTR_FRAME_COLOR, VAL_DK_GREEN);
520         SetCtrlAttribute (mainPanel, ctrlArrays[chip][3],
            ATTR_LABEL_TEXT, "Release Chip");
521         InstallCtrlCallback (mainPanel, ctrlArrays[chip][3],
            releaseChip, &instanceIndex[chip]);
522         ClearListCtrl (mainPanel, ctrlArrays[chip][6]);

523
524
525         currStep[chip] = 0;

            //reset step number
526         InsertListItem (mainPanel, ctrlArrays[chip][6], -1, stepNames[0
            ], 0);          //add to indicator list
527         loadState(stepRef[0][1], chip);
                                                          //load up first
            state
528         runActive[chip] = 1;
529
530         SetCtrlAttribute (mainPanel, ctrlArrays[chip][4], ATTR_ENABLED,
             1);            //start timer
531         ProcessSystemEvents();
532         return 0;
533     }
534
535 536 537
538     int CVICALLBACK resumeRun (int panelHandle, int controlID, int
        event, void *callbackData, int eventData1, int eventData2)
539     {
540         int* chipNo;
541         int chip;
542
543         if(event != EVENT_COMMIT) return 0;
544
545         chipNo = callbackData;
546         chip = *chipNo;
547
548
549         SetCtrlAttribute (mainPanel, ctrlArrays[chip][3],
            ATTR_LABEL_TEXT, "Release Chip");
550         InstallCtrlCallback (mainPanel, ctrlArrays[chip][3],
            releaseChip, &instanceIndex[chip]);
```

```
551         SetCtrlAttribute (mainPanel, ctrlArrays[chip][3], ATTR_DIMMED,
            1);
552
553         SetCtrlAttribute (mainPanel, ctrlArrays[chip][5], ATTR_VISIBLE,
             0);      // remove indication to add blood
554         SetCtrlAttribute (mainPanel, ctrlArrays[chip][0],
            ATTR_FRAME_COLOR, VAL_DK_GREEN);
555         runActive[chip] = 1;
556
557
558         currStep[chip]++;
                                                                          //load
            next state
559         InsertListItem (mainPanel, ctrlArrays[chip][6], -1, stepNames[
            currStep[chip]], currStep[chip]);         //add to indicator list
560         SetCtrlIndex (mainPanel, ctrlArrays[chip][6], currStep[chip]);
561         loadState(stepRef[currStep[chip]][1], chip);
562         SetCtrlAttribute (mainPanel, ctrlArrays[chip][4], ATTR_INTERVAL
            , (double)stepRef[currStep[chip]][2]);
563         SetCtrlAttribute (mainPanel, ctrlArrays[chip][1],
            ATTR_MAX_VALUE, stepRef[currStep[chip]][2]);
564         SetCtrlVal(mainPanel, ctrlArrays[chip][1], 0);
565         SetCtrlAttribute (mainPanel, ctrlArrays[chip][4], ATTR_ENABLED,
             1);
566
567         ProcessSystemEvents();
568         return 0;
569     }
570
571 572 573
574     int CVICALLBACK nextStep (int panel, int control, int event,
575               void *callbackData, int eventData1, int eventData2)
576     {
577         int* chipNo;
578         int chip;
579
580
581         if(event != EVENT_TIMER_TICK) return 0;
582
583         chipNo = callbackData;
584         chip = *chipNo;
585
586         if(runActive[chip] == 0) return 0;
587         currStep[chip]++;
```

```
        //increment current step
588 589
590     if(stepRef[currStep[chip]][0] == PAUSECODE)
                                        //pause should only occur when
        waiting for blood
591     {   SetCtrlAttribute (mainPanel, ctrlArrays[chip][5],
        ATTR_VISIBLE, 1);       // give indication to add blood
592         SetCtrlAttribute (mainPanel, ctrlArrays[chip][0],
            ATTR_FRAME_COLOR, VAL_RED);
593         SetCtrlAttribute (mainPanel, ctrlArrays[chip][3],
            ATTR_DIMMED, 0);
594         SetCtrlAttribute (mainPanel, ctrlArrays[chip][3],
            ATTR_LABEL_TEXT, "Continue");
595         InstallCtrlCallback (mainPanel, ctrlArrays[chip][3],
            resumeRun, &instanceIndex[chip]);
596         SetCtrlAttribute (mainPanel, ctrlArrays[chip][4],
            ATTR_ENABLED, 0);
597         InsertListItem (mainPanel, ctrlArrays[chip][6], -1,
            stepNames[currStep[chip]], currStep[chip]);          //add to
            indicator list
598         SetCtrlIndex (mainPanel, ctrlArrays[chip][6], currStep[chip
            ]);
599         runActive[chip] = 0;
600
601         return 0;
602     }
603
604     if(stepRef[currStep[chip]][0] == -1)
                                            //if run is over
605     {   SetCtrlAttribute (mainPanel, ctrlArrays[chip][0],
        ATTR_FRAME_COLOR, VAL_WHITE);
606         SetCtrlAttribute (mainPanel, ctrlArrays[chip][3],
            ATTR_DIMMED, 0);
607         SetCtrlAttribute (mainPanel, ctrlArrays[chip][3],
            ATTR_LABEL_TEXT, "Release Chip");
608         InstallCtrlCallback (mainPanel, ctrlArrays[chip][3],
            releaseChip, &instanceIndex[chip]);
609         runActive[chip] = 0;
610
611         return 0;
612     }
613
614
615     InsertListItem (mainPanel, ctrlArrays[chip][6], -1, stepNames[
        currStep[chip]], currStep[chip]);         //add to indicator list
616     SetCtrlIndex (mainPanel, ctrlArrays[chip][6], currStep[chip]);
```

```
617         loadState(stepRef[currStep[chip]][1], chip);
618         SetCtrlAttribute (mainPanel, ctrlArrays[chip][4], ATTR_INTERVAL
            , (double)stepRef[currStep[chip]][2]);
619         SetCtrlAttribute (mainPanel, ctrlArrays[chip][1],
            ATTR_MAX_VALUE, stepRef[currStep[chip]][2]);
620         SetCtrlVal(mainPanel, ctrlArrays[chip][1], 0);
621
622         ProcessSystemEvents();
623
624         return 0;
625     }
626
627 628 629
630
631     int CVICALLBACK updatePBars (int panel, int control, int event,
632               void *callbackData, int eventData1, int eventData2)
633     {
634         int i = 0;
635         int value;
636
637         if(event != EVENT_TIMER_TICK) return 0;
638
639         for(i = 0; i < MAXCHIPS; i++)
640         {    if(runActive[i])
641              {    GetCtrlVal(mainPanel,    ctrlArrays[i][1],    &value);
642                   SetCtrlVal(mainPanel,    ctrlArrays[i][1],    ++value);
643                   GetCtrlVal(mainPanel,    ctrlArrays[i][2],    &value);
644                   SetCtrlVal(mainPanel,    ctrlArrays[i][2],    ++value);
645              }
646         }
647
648         ProcessDrawEvents ();
649         return 0;
650     }
651
652 653 654
655 656 657
658 659 660
661     /***********************************
```

```
662  ********* MAPPING FUNCTIONS **********
663  ************************************/
664
665
666  int CVICALLBACK loadMapping (int panel, int control, int event,
667           void *callbackData, int eventData1, int eventData2)
668  {
669       char filePath[MAX_PATHNAME_LEN], stateFilePath[MAX_PATHNAME_LEN
         ];
670       char driveName[MAX_DRIVENAME_LEN], dirName[MAX_DIRNAME_LEN],
         fileName[MAX_FILENAME_LEN];
671       FILE* dataFile;
672       int status, i, j;
673       int currVal = 1;
674
675
676       if(event != EVENT_COMMIT) return 0;
677
678
679
680
681       status = FileSelectPopup ("", "*.smf", "*.smf", "Load mapping
         file", VAL_LOAD_BUTTON, 0, 1, 1, 1, filePath);
682       if(status < 1) return 0;
683
684       strncpy(stateFilePath, filePath, (strlen(filePath) - 3));
                              //copy into state file name, truncate
         extension
685       stateFilePath[strlen(filePath)-3] = '\0';
686       strcat(stateFilePath, "sta");
                                                          //add ".sta"
         extension
687       loadStateFile(stateFilePath);
688
689       loadMappingFromFile(filePath);
690
691       for(i = 0; i < MAXSOLENOIDS; i++)
                                                    //turn on all
         solenoids used for current device
692       {    if(solenoidLUT[currVal-1][i] != -1)
693            {    SetCtrlAttribute(pnlMapping, solenoids[solenoidLUT[
             currVal-1][i]], ATTR_ON_COLOR, VAL_DK_GREEN);
694                 SetCtrlVal(pnlMapping, solenoids[solenoidLUT[currVal-1
                 ][i]], 1);
695                 SetCtrlVal(pnlMapping, SCsolenoidConfig[i][0], 1);
696            }
697       }
```

```
698
699         for(j = currVal; j <= MAXCHIPS; j++)
700         {       for(i = 0; i < MAXSOLENOIDS; i++)
                                                  //turn on all solenoids
            used for current device
701             {   if(solenoidLUT[j-1][i] != -1)
702                 {   SetCtrlAttribute(pnlMapping, solenoids[solenoidLUT[
                    j-1][i]], ATTR_ON_COLOR, VAL_DK_GRAY);
703                     SetCtrlVal(pnlMapping, solenoids[solenoidLUT[j-1][i
                    ]], 1);
704                     SetCtrlAttribute (pnlMapping, SCsolenoidConfig[i][0
                    ], ATTR_ON_COLOR, VAL_DK_RED);
705                     SetCtrlVal(pnlMapping, SCsolenoidConfig[i][0], 1);
706                 }
707             }
708         }
709
710
711         activeSCsolenoid = -1;
712         activeABSsolenoid = -1;
713
714         return 0;
715     }
716
717 718
719     int CVICALLBACK saveMapping (int panel, int control, int event,
720             void *callbackData, int eventData1, int eventData2)
721     {
722         int i, j, k, totalSCsolenoids;
723         int status;
724         int currVal;
725         char tmpString[50], filePath[MAX_PATHNAME_LEN];
726         FILE* saveFile;
727
728
729         if(event != EVENT_COMMIT) return 0;
730
731         GetCtrlVal(pnlMapping, mapPanel_chipNumber, &currVal);
732         if(activeABSsolenoid != -1)
                                                              //make final
            assignment
733         {   solenoidLUT[currVal-1][activeSCsolenoid]        =
            activeABSsolenoid;            //    store in look up table
734             SetCtrlAttribute (pnlMapping, SCsolenoidConfig[
                activeSCsolenoid][0], ATTR_ON_COLOR, VAL_DK_RED); //
                darken to indicate
```

```
735             SetCtrlAttribute (pnlMapping, solenoids[activeABSsolenoid],
                 ATTR_ON_COLOR, VAL_DK_GREEN);            //indicate
                assignment made
736             activeABSsolenoid = -1;

737         }
738
739
740         //ADD ERROR CHECK TO ENSURE ALL SOLENOIDS ARE ASSIGNED!
741
742
743
744         status = FileSelectPopup ("", "*.smf", "*.smf", "Save mapping
            file", VAL_SAVE_BUTTON, 0, 1, 1, 1, filePath);
745         if(status < 1) return 0;
746         saveFile = fopen (filePath, "w");
747
                    //open file for write
748         fprintf(saveFile, "MAXSOLENOIDS: %i\n", MAXSOLENOIDS);
                                  //write array dimensions
749         fprintf(saveFile, "MAXCHPS: %i\n", MAXCHIPS);

750
751         for(i = 0; i < MAXCHIPS; i++)

752         {    for(j = 0; j < MAXSOLENOIDS; j++)
753                 fprintf(saveFile, "%i ", solenoidLUT[i][j]);
754             fprintf(saveFile, "\n");
755         }
756
757
758         fflush(saveFile); 759
            fclose(saveFile);
760
761
762         RemovePopup (pnlMapping);
763
764         return 0;
765     }
766
767 768
769     int CVICALLBACK cancelMapping (int panel, int control, int event,
770             void *callbackData, int eventData1, int eventData2)
771     {
772         if(event != EVENT_COMMIT) return 0;
773
```

```
774         RemovePopup (pnlMapping);
775
776         return 0;
777     }
778
779     int CVICALLBACK DecrementDevice (int panel, int control, int event,
780             void *callbackData, int eventData1, int eventData2)
781     {
782         int i;
783         int currVal;
784
785         if(event != EVENT_COMMIT) return 0;
786
787         GetCtrlVal(pnlMapping, mapPanel_chipNumber, &currVal);
788         if(currVal == 1) return 0;
789
790
791         if(activeABSsolenoid != -1)
                                                                    //make final
        assignment
792         {    solenoidLUT[currVal-1][activeSCsolenoid]         =
        activeABSsolenoid;             //     store in look up table
793             SetCtrlAttribute (pnlMapping, SCsolenoidConfig[
                activeSCsolenoid][0], ATTR_ON_COLOR, VAL_DK_RED); //
                darken to indicate
794             SetCtrlAttribute (pnlMapping, solenoids[activeABSsolenoid],
                 ATTR_ON_COLOR, VAL_DK_GREEN);             //indicate
                assignment made
795             activeABSsolenoid = -1;

796         }
797
798         for(i = 0; i < MAXSOLENOIDS; i++)
                                                        //turn off all
        solenoids used for current device
799         {    if(solenoidLUT[currVal-1][i] != -1)
800             {    SetCtrlAttribute(pnlMapping, solenoids[solenoidLUT[
                currVal-1][i]], ATTR_ON_COLOR, VAL_DK_GRAY);
801                 SetCtrlVal(pnlMapping, SCsolenoidConfig[i][0], 0);
802             }
803         }
804
805
806         SetCtrlVal(pnlMapping, mapPanel_chipNumber, (--currVal));
807
808         for(i = 0; i < MAXSOLENOIDS; i++)
                                                        //turn on all
```

```
        solenoids used for current device
809     {   if(solenoidLUT[currVal-1][i] != -1)
810         {   SetCtrlAttribute(pnlMapping, solenoids[solenoidLUT[
            currVal-1][i]], ATTR_ON_COLOR, VAL_DK_GREEN);
811             SetCtrlVal(pnlMapping, SCsolenoidConfig[i][0], 1);
812         }
813     }
814
815     activeSCsolenoid = -1;
816     activeABSsolenoid = -1;
817
818     return 0;
819 }
820
821 int CVICALLBACK IncrementDevice (int panel, int control, int event,
822         void *callbackData, int eventData1, int eventData2)
823 {
824     int i;
825     int currVal;
826
827     if(event != EVENT_COMMIT) return 0;
828
829     GetCtrlVal(pnlMapping, mapPanel_chipNumber, &currVal);
830     if(currVal == MAXCHIPS) return 0;
831
832
833     if(activeABSsolenoid != -1)
                                                        //make final
        assignment
834     {   solenoidLUT[currVal-1][activeSCsolenoid]        =
        activeABSsolenoid;            //    store in look up table
835         SetCtrlAttribute (pnlMapping, SCsolenoidConfig[
            activeSCsolenoid][0], ATTR_ON_COLOR, VAL_DK_RED); //
            darken to indicate
836         SetCtrlAttribute (pnlMapping, solenoids[activeABSsolenoid],
             ATTR_ON_COLOR, VAL_DK_GREEN);            //indicate
            assignment made
837         activeABSsolenoid = -1;

838     }
839
840     for(i = 0; i < MAXSOLENOIDS; i++)
                                                    //turn off all
        solenoids used for current device
841     {   if(solenoidLUT[currVal-1][i] != -1)
842         {   SetCtrlAttribute(pnlMapping, solenoids[solenoidLUT[
            currVal-1][i]], ATTR_ON_COLOR, VAL_DK_GRAY);
```

```
843                 SetCtrlVal(pnlMapping, SCsolenoidConfig[i][0], 0);
844             }
845         }
846
847 848
849         SetCtrlVal(pnlMapping, mapPanel_chipNumber, (++currVal));
850
851
852         for(i = 0; i < MAXSOLENOIDS; i++)
                                                        //turn on all
        solenoids used for current device
853         {   if(solenoidLUT[currVal-1][i] != -1)
854             {   SetCtrlAttribute(pnlMapping, solenoids[solenoidLUT[
                currVal-1][i]], ATTR_ON_COLOR, VAL_DK_GREEN);
855                 SetCtrlVal(pnlMapping, SCsolenoidConfig[i][0], 1);
856             }
857         }
858         activeSCsolenoid = -1;
859         activeABSsolenoid = -1;
860
861         return 0;
862     }
863
864 865
866     //SCsolenoid is the control IDs, x, y, theta pos for single-chip
        solenoid button, as defined in the user's state file
867     int CVICALLBACK toggleSCsolenoid (int panelHandle, int controlID,
        int event, void *callbackData, int eventData1, int eventData2)
868     {
869         int currChip = -1;
870         int i;
871
872         if(event != EVENT_LEFT_CLICK) return 0;
873
874
875
876         GetCtrlVal(pnlMapping, mapPanel_chipNumber, &currChip);
877         currChip--;
878
879         if(activeSCsolenoid != -1)
                                                                //if
        selecting a new SC solenoid
880         {   if(activeABSsolenoid != -1)
                                                            //   if an
        assignment was made
```

```
881             {    solenoidLUT[currChip][activeSCsolenoid] =
                activeABSsolenoid;            //     store in look up table
882                  SetCtrlAttribute (pnlMapping, SCsolenoidConfig[
                     activeSCsolenoid][0], ATTR_ON_COLOR, VAL_DK_RED); //
                     darken to indicate
883                  SetCtrlAttribute (pnlMapping, solenoids[
                     activeABSsolenoid], ATTR_ON_COLOR, VAL_DK_GREEN);
                              //indicate assignment made
884                  activeABSsolenoid = -1;

885             }
886             else
887             {    if(solenoidLUT[currChip][activeSCsolenoid] == -1)
888                       SetCtrlVal (pnlMapping, SCsolenoidConfig[
                          activeSCsolenoid][0], 0);              //turn off
                          previous solenoid
889                  else
890                       SetCtrlAttribute (pnlMapping, SCsolenoidConfig[
                          activeSCsolenoid][0], ATTR_ON_COLOR, VAL_DK_RED);
891             }
892        }
893
894        i=-1;

                  //locate & assign newly-active solenoid
895        while(controlID != SCsolenoidConfig[++i][0]);
896        activeSCsolenoid = i;
897        SetCtrlVal (pnlMapping, controlID, 1);
898        SetCtrlAttribute (pnlMapping, controlID, ATTR_ON_COLOR, VAL_RED
           );
899        activeABSsolenoid = solenoidLUT[currChip][activeSCsolenoid];
900        if(activeABSsolenoid != -1) SetCtrlAttribute (pnlMapping,
           solenoids[activeABSsolenoid], ATTR_ON_COLOR, VAL_GREEN);
901
902        return 0;
903   }
904
905   //solenoid[] is the control IDs for the absolutely-assigned 64
      solenoids that the single-chip versions will be mapped to
906   int CVICALLBACK toggleSolenoid (int panelHandle, int controlID, int
       event, void *callbackData, int eventData1, int eventData2)
907   {
908        int currChip = -1;
909        int currColor;

           //use to determine assigned state
910        int i, j;
```

```
911
912         if(event != EVENT_COMMIT) return 0;
913
914
915         GetCtrlVal(pnlMapping, mapPanel_chipNumber, &currChip);
916         currChip--;
917
918         if(activeSCsolenoid == -1)
919         {    GetCtrlVal(pnlMapping, controlID, &i);
920              SetCtrlVal(pnlMapping, controlID, 1-i);
921              return 0;
                 //no selecting if single-chip solenoid not selected
922         }
923         GetCtrlAttribute (pnlMapping, controlID, ATTR_ON_COLOR, &
            currColor);
924         if(currColor == VAL_DK_GRAY)
925         {    GetCtrlVal(pnlMapping, controlID, &i);
926              SetCtrlVal(pnlMapping, controlID, 1-i);
927              return 0;
                 //no selecting if solenoid is grayed out (used by another
                 chip)
928         }
929
930 931
932         if(currColor == VAL_DK_GREEN)
                                                              //if selected
            solenoid already associated in this chip
933         {    i = j = -1;
934              while(controlID != solenoids[++j]);
935              while(solenoidLUT[currChip][++i] != j);
                                                  // find associated
                 SCsolenoid in LUT
936              solenoidLUT[currChip][i] = -1;
                                                             // disassociate
937              SetCtrlVal(pnlMapping, SCsolenoidConfig[i][0], 0);
                                       // turn off SC solenoid
938              SetCtrlVal(pnlMapping, controlID, 0);
                                                    // and abs solenoid
939         }
940
941
942         if(activeABSsolenoid != -1)
943         {    SetCtrlVal (pnlMapping, solenoids[activeABSsolenoid], 0);
                //turn off previous selection
944              SetCtrlAttribute (pnlMapping, solenoids[activeABSsolenoid],
                  ATTR_ON_COLOR, VAL_GREEN);
```

```
945         }
946
947         i = -1;
948         while(controlID != solenoids[++i]);
                                                        //determine new
            active solenoid;
949         activeABSsolenoid = i;
950         SetCtrlAttribute (pnlMapping, solenoids[activeABSsolenoid],
            ATTR_ON_COLOR, VAL_GREEN);
951         SetCtrlVal (pnlMapping, controlID, 1);
                                                    //ensure new selection
            stays selected
952 953 954
955         return 0;
956     }
957
958
959     void setupMapping()
960     {
961         int i, j, row, col;
962         char lbl[5];
963         char fp[MAX_PATHNAME_LEN];
964         long fileSize;
965         int containerTop, containerLeft;
966         int ctrlTop, ctrlLeft;
967
968
969         int BTN_HEIGHT = 35,
970             BTN_WIDTH = 17,
971             rowSpacing = 12,
972             colSpacing = 25;
973         char callBack[20] = "toggleSolenoid";
974
975
976
977
978         for(i = 0; i < MAXCHIPS; i++)
979         {    for(j = 0; j < MAXSOLENOIDS; j++)
980             {    solenoidLUT[i][j] = -1;
981             }
982         }
983         activeSCsolenoid = -1;
984         activeABSsolenoid = -1;
985
986
```

```
987         GetCtrlAttribute (pnlMapping, mapPanel_dec_SolenoidContainer ,
            ATTR_TOP, &containerTop);
988         GetCtrlAttribute (pnlMapping, mapPanel_dec_SolenoidContainer ,
            ATTR_LEFT, &containerLeft);
989
990
991
992         for(i = 0; i < MAXSOLENOIDS; i++)
993         {    ctrlLeft = containerLeft + 10 + (colSpacing * (int)((i%32)/
            8)) + ((i%32) * (BTN_WIDTH + 5));
994              ctrlTop = (containerTop + 10 + ((int)(i/32) * (BTN_HEIGHT +
                  rowSpacing)));
995              sprintf(lbl, "%i", i+1);
996              solenoids[i] = NewCtrl (pnlMapping, CTRL_SQUARE_FLAT_BUTTON
                 , lbl, 500, 500);
997
998              SetCtrlAttribute (pnlMapping, solenoids[i], ATTR_ON_COLOR,
                 VAL_GREEN);
999              SetCtrlAttribute (pnlMapping, solenoids[i], ATTR_WIDTH,
                 BTN_WIDTH);
1000             SetCtrlAttribute (pnlMapping, solenoids[i], ATTR_HEIGHT,
                 BTN_HEIGHT);
1001             SetCtrlAttribute (pnlMapping, solenoids[i], ATTR_TOP,
                 ctrlTop);
1002             SetCtrlAttribute (pnlMapping, solenoids[i], ATTR_LEFT,
                 ctrlLeft);
1003             SetCtrlAttribute (pnlMapping, solenoids[i],
                 ATTR_LABEL_POINT_SIZE, 8);
1004             SetCtrlAttribute (pnlMapping, solenoids[i],
                 ATTR_LABEL_JUSTIFY, VAL_CENTER_JUSTIFIED);
1005             SetCtrlAttribute (pnlMapping, solenoids[i], ATTR_LABEL_LEFT
                 , ctrlLeft + (BTN_WIDTH/2) - 4);
1006             SetCtrlAttribute (pnlMapping, solenoids[i], ATTR_LABEL_TOP,
                  ctrlTop + (BTN_HEIGHT/2) - 5);
1007             SetCtrlAttribute (pnlMapping, solenoids[i],
                 ATTR_LABEL_BGCOLOR, VAL_TRANSPARENT);
1008             InstallCtrlCallback (pnlMapping, solenoids[i],
                 toggleSolenoid, NULL);
1009        }
1010
1011
1012
1013
1014        i = FileSelectPopup ("", "*.sta", "*.sta", "Load states file",
            VAL_LOAD_BUTTON, 0, 1, 1, 0, fp);
1015        if(i < 1) return;
1016        if(!(GetFileInfo (fp, &fileSize)))
```

```
                                                        //if file doesn't
        exist
1017    { MessagePopup ("Error", "There was a problem opening the
        file!");
1018        return;
1019    }
1020
1021    loadStateFile(fp);
1022
1023    return;
1024  }
1025
1026 1027
1028  int loadStateFile (char* filePath)
1029  {
1030      int i, j, k, status, bSize, numSolenoids;
1031      int varX, varY, flag;
1032      char readString[500];
1033      char stateName[30], junk[500];
1034      char volume[MAX_DRIVENAME_LEN], fileDir[MAX_DIRNAME_LEN],
        picName[MAX_FILENAME_LEN];
1035      char imageFileName[MAX_PATHNAME_LEN];
1036      char* token;
1037      FILE* dataFile;
1038      int BTN_HEIGHT = 30,
1039          BTN_WIDTH = 15;
1040      int xCorrect = 350,
1041          yCorrect = -25;
1042      float scaler = 0.78125;
1043
1044
1045
1046      dataFile = fopen (filePath, "r");
1047
1048      //load image file
1049      fgets (readString, sizeof(readString), dataFile);
1050          SplitPath (filePath, volume, fileDir, NULL);
                                              //get current directory
1051          for(i = 9; i < strlen(readString); i++)
                                                  //get
          filename

1052              picName[i-9] = readString[i];
1053              picName[i-10] = '\0';
```

```
1054            if(strlen(picName) != 0)
1055            {   sprintf(imageFileName, "%s%s%s", volume, fileDir,
                picName);           //construct picture path
1056                DisplayImageFile (pnlMapping, mapPanel_pctUserImg,
                    imageFileName);
1057            }
1058
1059
1060        //create user-placed solenoid buttons
1061        fgets (readString, sizeof(readString), dataFile);
1062            sscanf(readString, "solenoids: %i %i\n", &numSolenoids, &
                bSize);
1063            for(i = 0; i < numSolenoids; i++)
1064            {   SCsolenoidConfig[i][0] = NewCtrl (pnlMapping,
                CTRL_SQUARE_LED, "", 100, 100);           //create in a pile
                for now
1065                SetCtrlAttribute (pnlMapping, SCsolenoidConfig[i][0],
                    ATTR_ZPLANE_POSITION, 100);      //hide behind picture
1066                SetCtrlAttribute (pnlMapping, SCsolenoidConfig[i][0],
                    ATTR_HEIGHT, BTN_HEIGHT);
1067                SetCtrlAttribute (pnlMapping, SCsolenoidConfig[i][0],
                    ATTR_WIDTH, BTN_WIDTH);
1068                InstallCtrlCallback (pnlMapping, SCsolenoidConfig[i][0
                    ], toggleSCsolenoid, NULL);
1069            }
1070
1071        //scale & place buttons on image
1072        for(i = 0; i < numSolenoids; i++)

                //                  right click)
1073        {   fgets (readString, sizeof(readString), dataFile);
1074            sscanf(readString, "%i %i %i", &SCsolenoidConfig[i][1], &
                SCsolenoidConfig[i][2], &flag);
1075            SCsolenoidConfig[i][1] -= xCorrect;       SCsolenoidConfig[i][1
                ] *= scaler;
1076            SCsolenoidConfig[i][2] -= yCorrect;       SCsolenoidConfig[i][2
                ] *= scaler;
1077            if(SCsolenoidConfig[i][1] != -1) SetCtrlAttribute(
                pnlMapping, SCsolenoidConfig[i][0], ATTR_TOP,
                SCsolenoidConfig[i][2]);
1078            if(SCsolenoidConfig[i][2] != -1) SetCtrlAttribute(
                pnlMapping, SCsolenoidConfig[i][0], ATTR_LEFT,
                SCsolenoidConfig[i][1]);
1079            if(SCsolenoidConfig[i][2] != -1) SetCtrlAttribute (
                pnlMapping, SCsolenoidConfig[i][0], ATTR_ZPLANE_POSITION, 0
                );
1080            if(!flag)
```

```
1081            {   SetCtrlAttribute (pnlMapping, SCsolenoidConfig[i][0],
                ATTR_HEIGHT, BTN_WIDTH);
1082                SetCtrlAttribute (pnlMapping, SCsolenoidConfig[i][0],
                    ATTR_WIDTH, BTN_HEIGHT);
1083            }
1084            SetCtrlAttribute (pnlMapping, SCsolenoidConfig[i][0],
                ATTR_ON_COLOR, VAL_GREEN);
1085        }
1086
1087
1088        return 0;
1089    }
1090
1091 1092
1093 1094
```