ABSTRACT

Title of thesis: DESIGN FOR A STAND-ALONE, UNIVERSAL SERIAL BUS (USB) -ENABLED AIRFLOW PERTURBATION DEVICE

Nischom Karl Silverman, Master of Science, 2004

Thesis directed by: Professor Arthur T. Johnson
Biological Resources Engineering Department

The stand-alone, USB-enabled Airflow Perturbation Device (APD) provides an average respiratory resistance (RR) measurement without connection to a computer and offers expanded functionality when connected to a computer. In both home and medical clinic settings, RR can provide a measure of impairment in obstructive respiratory disorders and the effectiveness of respiratory therapies. The APD measures RR during passive breathing by sensing the ratio of pressure increase to flow reduction during brief, partial airflow interruptions. Prior work has shown the APD to produce repeatable, sensitive RR measurements in humans and animals. The device of prior investigations incorporated a computer and data acquisition card. The research presented here demonstrates that the APD can provide accurate measurements in a stand-alone format and provide expanded function with a USB host computer.

DESIGN FOR A STAND-ALONE, UNIVERSAL SERIAL BUS (USB) -ENABLED AIRFLOW PERTURBATION DEVICE

by

Nischom Karl Silverman

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland at College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2004

Advisory Committee:

Professor Arthur T. Johnson, Chair
Associate Professor Hubert J. Montas
Professor Yang Tao

©Copyright by

Nischom Karl Silverman

2004

DEDICATION

This work is dedicated to my family who has supported my education

and encouraged in me a sense of its true values.

ACKNOWLEDGMENTS

The author is grateful for the energies of many people without whose invaluable support this work could not have been completed. In particular the author wishes to thank:

Dr. Arthur T. Johnson for his guidance, his perspective on engineering concepts, and his persistence in voicing his belief that the author was capable of completing this project,

Dr. Hubert J. Montas for his dedication to student support, helpful ideas, practical suggestions, and perspective

Dr. Yang Tao for his support, criticism, practical guidance, and insight into industrial perspective

Dr. John Jeka for his guidance and support,

Dr. Adel Shirmohammadi for his student advocacy,

all Biological Resources Engineering and Kinesiology faculty, staff and graduate students for their support in daily matters, comraderie, perspectives and suggestions,

the Internet hardware and software developer community for the energy devoted to user forums and the freedom of information and tools,

all colleagues, coworkers and friends elsewhere for their patience and acceptance each time the author had to forgo an opportunity, and their assurance that there would be future opportunities at the completion of this work,

all loved ones in my life for their encouragement and illumination when the light at the end of the tunnel was not yet in sight.

TABLE OF CONTENTS

LIST OF FIGURES....vi

LIST OF TABLES....viii

LIST OF ABBREVIATIONS....ix

INTRODUCTION....1
- Respiratory Measurement Device Needs: Homes and Medical Clinics....1
- Research Goal....4

LITERATURE REVIEW....6
- Prevalence of Pulmonary Disorders....7
- Where and When Respiratory Function Should Be Assessed....8
- What Characteristics a Measurement Device Can Assess....10
  - Respiratory Mechanical Properties....11
  - Respiratory Performance Characteristics....13
- The Impact of Measurement Technique on Respiratory Function....14
- Measurement Techniques....16
  - Physical Structure Imaging....16
  - Lumped Characteristic Measurement....17
    - Acoustic Analysis....17
    - Spirometric Assessment....18
    - Esophageal Balloon....23
    - Full Body Plethysmography....24
    - Interrupter Devices....25
    - Forced Oscillation Technique: Pressure Addition....26
    - Perturbation Techniques and the APD: Flow Reduction....27

OBJECTIVES....29
- Design Specifications....29

EQUIPMENT....34
- APD Design Prior to this Research: APD100....34
- Equipment Available: Options for APD Design Revisions....36
- Equipment: Revised Design for APD-SA USB....36
  - APD-SA Components....38
  - APD-SA Operation....44
    - Division of Functionality....44
    - Firmware Descriptions....46
    - Software Description....52

PROCEDURES....55
PC Hosted Prototype Development....55
Integrated Hardware and Software Development....56
APD-SA Performance Validation....59
Calibration Performance....59
Human Subject Testing....60
Further Investigation of RR Calculation Algorithms....61

RESULTS AND DISCUSSION....63
Design Results....63
Calibration Results....68
Subject Testing Results....73
Subject Summary Data....73
APD-SA and APD100 Average RR Comparison....74
Investigation of Measurement Differences....76

CONCLUSIONS....80

SUGGESTIONS FOR FURTHER STUDY....81
Explore APD Algorithms....81
Improve APD Hardware and Software....82
APD System Expansion....82
APD Software....83
APD100 Software....84
APD200 Software....85
APD-SA Performance in Firmware....86
APD-SA Hardware Improvements....88
APD Body Mechanical Improvements....89

APPENDICES....90

REFERENCES....336

LIST OF FIGURES

Figure 1. Block diagram of the design for the APD-SA USB ... 37

Figure 2. A rendering of the APD-SA in which the APD-SA enclosure housing electronics mounts onto the APD body that is also used in the APD100 system ... 39

Figure 3. APD-SA ADC and logic schematic for analog signal conversion, motor control and interfacing APD-SA PCBs; the figure includes all headers for connections between PCBs; all electrical connections, or nets, are named, and identical names indicate electrical connection, even if a trace is not drawn between the named points; the PCB layout was derived from this complete schematic ... 42

Figure 4. Interconnections and data exchange among the ICs and PCBs in the APD-SA enclosure ... 43

Figure 5. Interrupt driven communication among APD-SA DSP, USB and host PC ... 45

Figure 6. Diagram of the APD-SA EZUSB firmware; the polling loop (top) executes continuously; if a command is received from the USB host PC, then the EZUSB executes the necessary actions to comply with the command (bottom) ... 47

Figure 7. Diagram of APD-SA DSP firmware operations upon power-up and the point at which the DSP will heed PC-initiated USB commands ... 48

Figure 8. Diagram of the APD-SA DSP firmware polling loop that toggles among operating modes and enables timer interrupt routines to stream raw data or measure RR ... 50

Figure 9. Diagram of the APD-SA TimeProcRunMode timer interrupt routine that triggers at the sample rate to measure pressure and flow to calculate RR ... 51

Figure 10. Diagram of the APD-SA implementation of the EZUSB "Renumeration" feature when APD200 invokes USB communication; USB cable connection (top left) and APD200 program execution (top right) are independent of each other and can occur in any order; APD200 software has provisions for both the presence and absence of the APD-SA USB connection ... 54

Figure 11. APD-SA USB ready to make an RR measurement ... 63

Figure 12. A subject using the APD-SA USB as a stand-alone device to measure his RR....64

Figure 13. LCD display sequence on the APD-SA USB; clockwise from the top left: two start-up screens followed by screens that indicate to the user that the device is reading data; “Read:” followed by a percentage indicates to the user how much of the measurement has been completed, as a percentage of the total number of required good perturbations; the dashes at the bottom of the “Read:” screen move from left to right with every few good perturbations and the displayed percentage increases in 10% intervals; when a sufficient number of perturbations have been acquired, the screen displays the average RR; at any time during operation, a USB connection to the device will be indicated with the “USB” screen....65

Figure 14. APD200 main panel window with menu functions and RR data display, shown here in operation with the APD-SA USB device....66

Figure 15. APD200 window that indicates to the user the status of the connection to the APD-SA USB when a USB connection to the device is first attempted....66

Figure 16. APD200 window that shows calibration constants, shown here in operation with the APD-SA USB; the window includes buttons to transfer calibration constants to and from the APD-SA USB....67

Figure 17. APD200 window that shows a plot of individual RR values over time measured by the APD-SA USB....68

Figure 18. APD-SA average RR plotted against APD100 average RR for 13 subjects....75

Figure 19. Device resistance, pressure, and flow during an inhalation (negative flow and pressure) APD perturbation; APD100 samples every several data points until one is found above an RD threshold (RD, thres); when the first point is found (A), APD100 then scans backwards for the first point at which RD is below the threshold and the slope of RD is negative (B); APD100 then scans forward for the first point having the same value and slope characteristics as (B) which is (C); RR is then calculated based on flow slope (m, flow) and pressure slope (m, pressure) between points (B) and (C); true RR would probably best be calculated between (C) and (D) or (B) and (E)....79

LIST OF TABLES

Table 1. Selected commercially available peak flow meter (PFM) and pulmonary function test (PFT) device models 21

Table 2. Summary information for the four printed circuit boards in the APD-SA enclosure 40

Table 3. Span calibration results for APD100 and APD-SA; APD100 collects and scales voltage data while the APD-SA collects and scales ADC counts 69

Table 4. Data for a comparison of calibration results for APD100 and APD-SA, showing the measured and theoretical spans and percent difference between theoretical and measured values for each span; pressure to flow span ratio is the pressure span divided by flow span, the ratio that converts an unscaled respiratory resistance value to a scaled value 71

Table 5. Specifications relevant to calibration span for sensing and data conversion components on the APD100 and APD-SA 71

LIST OF ABBREVIATIONS

AC	alternating current
ADC	analog to digital converter
ANSI	American National Standards Institute
APD	Airflow Perturbation Device
APD100	APD sensor and data processing unit prior to this research that requires data acquisition card and computer
APD200	APD host computer software that communicates with APD-SA over USB and includes APD100 functionality for backward compatibility
APD-SA	stand-alone APD designed in this research
APD-SA USB	refers to APD-SA but highlights its USB functionality
API	application programming interface – functions that hook into operating system or driver capabilities
BRE-UMCP	Biological Resources Engineering Department of University of Maryland at College Park
CAD	computer aided drafting
cmH2O	measure of pressure as centimeters of water
COPD	chronic obstructive pulmonary disease
CPU	central processing unit
DAQ	data acquisition card
DB9-F	D-subminiature 9-pin connector, female
DB9-M	D-subminiature 9-pin connector, male
DC	direct current
DRD	derivative of the device resistance
DSP	digital signal processor; a CPU/ MCU specifically optimized for typical digital signal processing data operations
EEPROM	electrical erasable programmable read only memory
Exh	exhalation direction of breathing cycle
EZUSB	USB controller chip based upon 8051 MCU architecture released by Cypress semiconductor – especially part number AN2131Q
FBP	full body plethysmography
FEV1	forced expiratory volume in one second
FLASH	a very fast type of EEPROM
FO	forced oscillation
FVC	forced vital capacity
HID	human interface device – a class of USB device
Hz	Hertz, cycles per second
I2C	two-wire bi-directional serial communication interface used especially for chip-to-chip communication on PCBs

IC	integrated circuit
IDE	integrated development environment
Inh	inhalation
inH2O	measure of pressure as inches of water
IO	input/ output; communications in and out of a device
JTAG	Joint Test Action Group – refers to a standardized protocol created by this group for in-circuit firmware downloading and debugging in embedded processors
LCD	liquid crystal display
Lps	measure of flow rate as Liters per second
mA	milliamps
MCU	microcontroller
OS	operating system
PC	personal computer
PC104	standardized format for small footprint single board computers functionally resembling personal computer motherboards
PCB	printed circuit board
PEF	peak expiratory flow
PFM	peak flow meter
PFT	pulmonary function test
RAM	random access memory
RAW	resistance of the airways
RD	resistance of the device
RMD	respiratory monitoring device
ROM	read-only memory
RR	respiratory resistance
RS232/485	serial communication protocol common to PC
SRAM	static RAM
USB	universal serial bus; high-speed serial communication protocol
V	volts
VAC	volts – specifically alternating current volts
VB	Microsoft Visual Basic (6.0)
VDC	volts – specifically direct current volts
W	watts

INTRODUCTION

The Airflow Perturbation Device (APD) non-invasively measures in humans and other animals respiratory resistance (RR) during passive, spontaneous breathing (Lausted and Johnson, 1998; Johnson et al., 1984a). The APD is useful because RR measurements and other measures of respiratory system properties allow the assessment, in a population, of average respiratory characteristics and the degree to which a disease or environmental condition has altered these characteristics.

Respiratory Measurement Device Needs: Homes and Medical Clinics

Respiratory system-altering disorders and conditions occur frequently. For example, approximately 17 million people, or six percent of the United States population, have airways-narrowing asthmatic disorders (United States Centers for Disease Control, 1998). This prevalence creates a need for respiratory measurement devices in clinics, ambulances, and hospitals for assessment and monitoring. Home recovery, rehabilitation, and monitoring programs often supplement clinical treatment and augment the need for respiratory assessment devices, especially inexpensive, small and easy-to-use devices.

Many devices partially satisfy the need for respiratory measurement in clinical and home settings. Body plethysmography and spirometric pulmonary function testing

prevail in hospitals and clinics, respectively, and peak flow monitoring dominates the home respiratory measurement device market. Though they provide useful information, each of these measurement methods suffers drawbacks in its ease-of-use, expense, accuracy, or utility for home-use obstructive respiratory disease monitoring. The APD offers advantages to these methods.

Further, the respiratory system is dynamic and common measurement methods fail to easily depict the short-term changes in respiratory function. Some changes in measurable properties occur during a single breath. Some changes occur on the order of a minute, whether normal or disease-induced. Some changes occur over several years as part of the aging process or as part of the progression of lung damage. Thus, fast, continuous, sensitive measurements of respiratory function can illuminate any decrement in respiratory response for those respiratory characteristics that have a time component. Measurement accuracy and repeatability are important for long-term changes, while quick-response is important for short-term changes. The APD's sensitivity and repeatability has been demonstrated in Lausted (1997) and Silverman et al. (2002) have demonstrated its ability to show short term airways caliber changes.

Regardless of the integrity of the measurement method, the form that a respiratory measurement device (RMD) takes defines its cost, ease of use and thereby the extent of its use. Most measurement methods targeted for home-use measure conglomerate characteristics of the respiratory system, like RR or peak exhalation flow, and require no technicians or special auxiliary apparatus like computers or sealed chambers. These qualities make reasonable the widespread use of peak flow as a home-use RMD. In its simplest form, the device requires only mechanical parts constructed

such that when a patient blows into the device, an indicator moves along a scale in proportion to the peak flow passed through the device, enabling a cheap, disposable plastic construction. Other spirometric methods such as continually monitored forced exhalation require a computer for detailed data analysis and are restricted to a laboratory. However, some peak flow meters and spirometric devices are available for home use that incorporate electronics for more precise measurements or more detailed information. Their price increases according to their functionality and complexity, defined by the flow sensing, signal processing and display hardware, as well as their mechanical packaging.

The home-use spirometric products on the market most likely represent final products built upon concepts from tested prototypes. The custom RMD housing and custom circuitry optimized for minimal space and cost represent a large initial capital investment in time and materials, for example in the tooling of machines. Building a prototype RMD in the Human Performance Laboratory, Biological Resources Engineering, University of Maryland at College Park (HPL, BRE-UMCP) defines a different set of constraints – those restricting the initial capital investment and project focus. Though some of the capital may be affordable despite its expense, the time required to work with the specialized equipment must also be considered. Thus, the project discussed here will be considered an endeavor to prototype a home-use APD, using materials that may cost more per item purchased, but require less capital investment. This will allow the evaluation of the design and data processing methods, leaving refinement of circuitry and housing to future steps in product development. As

such, the final product of this endeavor will be a product that is testable in ease of use and reliability and can be easily used in laboratories, clinics or at home.

Research Goal

The objective of this research is to continue the development of the APD into a device that demonstrates the APD's suitability for clinical and home use. The development is intended to produce a form of the APD that incorporates similar measurement methods to those in Lausted (1997), but resides in a more portable unit fully dedicated to APD RR measurement. In this development, the human-device interface is now limited to system on-off and measurement start-stop actions most readily. Calibration functions and more detailed subject data will be available upon connection to a host personal computer (PC), typically in a clinical or similar advanced user setting.

The stand-alone form is built around microcontroller (MCU) development platforms intended for embedded and hand held applications with custom data collection electronic hardware. The custom display, input-output (IO), user-interface and data collection hardware eliminates the costs of a data acquisition card (DAQ) typically $400 to $500 and makes optional a PC. Prior to this research the APD required a DAQ and PC for operation. Given the state-of-the-art technology readily

available, this APD revision might have taken many forms, but uses hardware that most appropriately meets the development needs.

LITERATURE REVIEW

The review of the literature demonstrates that a modified APD that can be used in clinics or at home is needed because:

1) Obstructive respiratory disorders that often require or benefit from home respiratory monitoring occur frequently.
2) Home-use respiratory monitoring requires simple, inexpensive devices.
3) Home-use as well as clinical respiratory monitoring is presently dominated by devices that suffer drawbacks such as indirect measurement of airways obstruction, effort dependency and poor sensitivity.
4) There are many techniques that can measure airways obstruction more directly as airways or respiratory resistance.
5) Most techniques that can measure airways obstruction more directly are not suitable for home-use.
6) The APD measures respiratory resistance and is suitable for home and clinical use, but its configuration prior to this research confines it to the laboratory.

Prevalence of Pulmonary Disorders

Pulmonary disorders comprise a great portion of medical focus. In the United States, approximately 14 million citizens require treatment for chronic obstructive pulmonary disorders (COPD), such as emphysema and chronic bronchitis (American Thoracic Society, 1998). In 1993, asthma and COPD induced over 17 million physician office visits per year, at a cost of about 10.4 billion dollars (Higgins, 1993). In 1999, there were over 190,000 asthma-induced hospitalizations for children younger than 15 years of age, and near 500,000 hospitalizations in the United States (National Heart, Lung and Blood Institute, 2002, 1997). The impact of asthma on our society has been increasing rapidly since 1980 as measured by the number of diagnoses, doctor visits, hospitalizations and missed work or school days (American Lung Association, 2001). Various other respiratory disorders demand medical attention, including pneumonia, acute bronchitis, pulmonary embolism, laryngeal dyskinesia, interstitial disorder, lung cancer, post-operative pulmonary complications, and pulmonary allergic reactions.

Some respiratory system impairment is known to occur directly from environmental conditions. Wong et al. (1998) found that air pollution induced bronchial hyperresponsiveness. Obase (1999) examined in workers the nature of bronchoconstrictive asthma attacks induced by buckwheat and wheat flour inhalation. Mazumdera (2000) found that the presence and degree of arsenic poisoning could be determined by an examination of the degree of respiratory impairment. Thus,

respiratory impairment resulting from environmental pollutant exposure contributes greatly to the number of respiratory-related medical cases.

Where and When Respiratory Function Should Be Assessed

Successful study, diagnosis, treatment, and rehabilitation for pulmonary disorders require an accurate battery of pulmonary function tests. Extensive respiratory function tests must take place in hospitals and outpatient clinics to diagnose and monitor acute or chronic respiratory disorders as well as monitor respiratory function that has been impaired from a treatment intended to address a non-respiratory ailment. Many hospitalizations require respiratory monitoring in addition to other monitors following treatment. For example, Snowden et al. (2000) found respiratory edema, the accumulation of fluids in respiratory tissues, to be a common complication in post-operative liver transplant patients. Such post-operative conditions increase the workload for the respiratory system and hamper recovery. Pulmonary function tests, such as a battery of forced exhalation maneuvers, have historically created a baseline or common measure for studying respiratory diseases or monitoring patients during in-patient recovery (Gern et al., 1997; Trigg et al., 1996; Wojnarowski et al., 1997; Martinez et al., 1990; Jacob et al., 1997). Asthma patients including children are encouraged to perform with a peak flow meter (PFM) PEF tests each day to monitor their ever-changing bronchoconstriction as part of a home monitoring program.

Simpler pulmonary function tests must also be available in homes to facilitate home-care rehabilitation programs or daily monitoring. Hernandez et al. (2000) found that home-based care for COPD patients proved as successful as in-patient treatment. Home-care pulmonary rehabilitation programs have been found to maintain improved pulmonary health over a longer time period than outpatient rehabilitation clinics (Goldstein et al., 1994; Wijkstra et al., 1996). Dirksen et al. (1998) recommend continuous respiratory health monitoring for COPD patients every several months because changes in respiratory function in these cases occur on this time scale and necessitate continual evaluation. Vollmer et al. (2002) found that quality of life significantly improves for even mild asthma patients when they closely monitored and regularly treated their condition. Eid et al. (2000) suggest that monitoring respiratory health several times daily at home is an essential component to asthma management.

Simple home monitoring tests facilitate continuous monitoring by reducing the need to visit a clinic and making each measurement easy. A device that consistently provides a valuable measure of respiratory characteristics in homes and hospitals satisfies medical professionals' and respiratory patients' needs for diagnosis, treatment, and rehabilitation. A device that provides respiratory information in the ambulance, hospital, and home with equal accuracy would serve to standardize respiratory health measures and ease comparison of data in and out of health care institutions.

What Characteristics a Measurement Device Can Assess

Respiratory disorders alter the performance of the respiratory system, often in a measurable manner. For example, smoking-induced emphysema produces permanent alveolar fusion and reduced oxygen transfer surface area in the lungs. Measuring blood dissolved gas levels might provide an estimate of reduced oxygen transport. Some respiratory diseases such as pulmonary dysplasia, pulmonary embolism, interstitial disorder and lung cancer can be examined in their early stages by scanning the respiratory system's physical structure for abnormalities via imaging techniques. Many respiratory afflictions impact respiratory mechanical characteristics such as lung capacity, respiratory tract elasticity, breathing cycle characteristics, and energy required to maintain respiration. Bronchial pathway restriction, air trapping outside the lungs, and edema also characterize emphysema (Verbeken et al., 1996). These changes are reflected in altered respiratory mechanical properties.

Of particular relevance to the APD are those properties of the respiratory system that can be measured as a lumped characteristic for each subject and quickly compared with expected values. Such lumped characteristics are either average mechanical properties of the respiratory system, or average respiratory performance. RMDs have been designed to measure the first or the latter. In respiratory clinics and at home, respiratory performance is often used as an indirect indicator of respiratory mechanical properties, though it is respiratory performance that without instrumentation may ultimately alert each patient to his or her troubles.

Respiratory Mechanical Properties

The mechanical properties of the respiratory system are resistance, compliance, and inertance. Lumped values of each are often assigned to each of three portions of the respiratory system: airways, lung tissue and chest wall. Average values over two or more of these portions is often what is measured in practice. The properties characterize the manner in which a difference in pressure between the base of the lungs and atmosphere cause airflow. Resistance occurs in the respiratory system as an opposition to airflow in the upper and lower airways, resistance to movement in the lung tissue, and a resistance to movement in the chest wall (Johnson, 1991). Compliance is the change in pressure due to an induced volume change, representing elastic properties of the respiratory system. Inertance is the tendency for air and tissue to resist changes in speed of movement. Inertance exerts greater effect for larger changes in speed of movement and larger mass. Both compliance and inertance are frequency dependent properties. In respiratory measurements, the frequency component arises from the breathing frequency or the frequency at which the respiratory system is excited during a measurement.

In the context of the APD, inertance and compliance are less important than RR for several reasons. APD measurements excite the respiratory system between six and ten cycles per second, near the respiratory system's resonance frequency (Lehtola, 1986). At this frequency, compliance and inertance become very small. Barnas et al. (1989) found the chest wall component of compliance to decrease slightly as the chest wall was moved incrementally faster from 3 Hz to 10 Hz, and to increase with larger

tidal volumes, the volume of a breath, at very low excitation frequencies. This further suggests that compliance is low near the APD's operating frequency. Lausted (1997) found compliance to be very small - within the noise range – for APD measurements. Inertance is generally neglected in respiratory measurements (Johnson, 1991). The APD minimally changes airflow, further supporting the neglect of inertance. Thus RR remains the dominant measurable parameter for consideration in this research.

Numerous studies have been conducted to examine RR throughout the breathing cycle and to observe the effect of different breathing and periodic excitation rates on RR. This information has, for the most part, been obtained through invasive techniques to measure the pressure within the body that creates the pressure gradient to drive flow. For example, at the end of expiration, RR increases as smaller airways collapse and, in a forced exhalation, respiratory muscles contract to force air from the lungs. Properties of the respiratory system have been measured while researchers induced air and respiratory tissue motion at various cyclic rates. Research results indicate that the chest wall component of RR has a minimum value between about 3 Hz and 10 Hz excitation frequencies (Bouhuys and Jonson, 1967; Barnas, et al., 1990). For low excitation frequencies, chest wall resistance was found to be strongly effected by the degree of respiratory muscle contraction, but the effect diminished at higher frequencies, near 4 Hz (Barnas et al., 1989). This information exemplifies the time component in some of the measurable properties of the respiratory system. It also suggests that a respiratory system might be characterized as impaired based on its short time-scale behavior, in a sense fingerprinting a respiratory system and subsequently determining if that fingerprint resembles that of an impaired respiratory system.

Respiratory system changes occur on slightly longer time-scales as well, some time-scales on the order of years. Moderate to severe asthma causes increased lung volumes and airflow restrictions that vary on an hourly timescale (Eid et al., 2000). Silverman et al. (2002) have found that RR is decreased just after moderate exercise and increases towards pre-exercise values within about one minute. Tidal COPD airflow changes vary only over long time periods (Cosio and Guerassimov, 1999; Celli, 1995).

Respiratory Performance Characteristics

Many respiratory assessments are made by inferring the mechanical properties of the respiratory system from respiratory performance. Pulmonary function tests (PFTs) such as spirometry measure the peak flow rates that a subject's respiratory system can generate. This provides an indirect estimate of the degree of bronchoconstriction, because RR is correlated with resistance of the airways (RAW) (Ducharme, et al., 1998). PFTs also provide a measure of diaphragm strength and tissue elastance. Performance tests can also directly measure some physiological characteristics of the respiratory system as well. If a maximum inhalation is followed by a maximum exhalation, then the respiratory capacity can be measured.

The Impact of Measurement Technique on Respiratory Function

It seems clear that to characterize a system most accurately, a measurement should minimize the degree to which the system must change to facilitate the measurement. Otherwise, the measurement yields not the intended information about the system characteristic in its current state, but information about the system when it has changed to accommodate a measurement. For respiratory system characterization, the act of measuring a respiratory system that strains to make the measurement has achieved a measurement of the strained respiratory system and any of the resulting changes that the strain immediately or soon thereafter induces. The impact of the measurement method upon the measured system should be considered, including the physiological and psychosomatic effects.

Several respiratory ailments have strong cognitive and perceptual components. Dyspnea, the general sensation of improper respiratory functioning, and laryngeal dyskinesia include true physical components but are strongly influenced by patients' perception (American Thoracic Society, 1998; Carrieri-Kohlman et al., 1996). Providing patients with actual physical descriptions of their respiratory performance can aid in alleviating excessive affective disorder components (Ramirez, 1986).

Some measurements may actually induce affective changes in the patient. It is possible that performance anxiety may result from maneuvers in which the patient is stressed to make a measurement. Such measures include forced exhalation in which a patient is forcefully coached to expel as much air as possible. Affective components of

dyspnea imply that the perception of effort varies with external conditions (American Thoracic Society, 1998). Hence a patient may fail to consistently apply the same maximal effort in spirometric testing. The inconsistency of effort-dependent measures including the forced exhalation maneuver has been frequently cited but is often neglected (Goldsmith, 2002). Given repeated trials and familiarity with the testing procedure, patients may be able to use a portable spirometric device at home. In the literature, the numerous studies based upon forced exhalation PFTs testify to the degree to which the effort dependence is not considered to be a great hindrance.

Invasive measures in which a patient must allow a measurement device into his or her body might cause stress to a subject until he or she becomes acclimated to the device. An example of invasive measurement is the esophageal balloon technique in which a subject swallows a small balloon attached to a tube to indicate pressure near the base of the lungs.

Beyond affective patient changes, some test methods may physiologically alter the respiratory system under study. Since lung volume and flow rate effect respiratory system properties, the measurement technique should minimize the need for special breathing patterns to achieve a measurement or at least track the changes during breathing if highly precise measurements are desired throughout the breathing cycle. Spirometry requires deep inhalation and exhalation. Spirometric breathing maneuvers may introduce error since deep inhalation prior to maximal exhalation can cause transient bronchodilation (Widdicombe, 1989; Burns, et al., 1985). Extreme forced exhalation required for spirometry can cause bronchoconstriction or reduction in vital capacity in asthmatics (Herxheimer, 1975). Some body plethysmography techniques

require hyperventilatory panting maneuvers and hence altered respiration rates. Further, if long-term body plethysmography measurements are made, gas composition must be adjusted to avoid gas composition-induced respiratory changes that occur as gas sensors in the body adjust respiration to compensate for changes in carbon dioxide and oxygen concentrations.

Measurement Techniques

Several techniques are used to measure respiratory properties in laboratories, hospitals, clinics and at home. Each is restricted in the information that it provides. Many present limitations to their widespread home or clinical use.

Physical Structure Imaging

Several assessment techniques provide relevant respiratory system data. Computed tomography (Wu et al., 1994) provides a map of pulmonary physical structure and radiology (Snowden et al., 2000) provides similar maps in less detail. However, these time-consuming methods demand space-consuming scanning and interpretation apparatus and experienced technicians, prohibiting their more widespread implementation. Their value lies in detailed physical analysis.

Lumped Characteristic Measurement

Lausted (1997) and Johnson et al. (1984a,b) provide reviews of respiratory characteristic measurement methods and highlights aspects of each that are particularly relevant to an assessment of APD utility. A summary of respiratory measurement techniques discussed therein is presented here with additional information on new developments and the application of these techniques to home and clinical use. A more complete description of respiratory measurement techniques not covered in the reviews but relevant to home use is also included here.

Acoustic Analysis

In acoustic analysis, researchers attempt to correlate breathing sounds and respiratory condition. Microphones such as contact sensors can be placed on the chest to record breathing sounds that are analyzed for characteristics such as frequency content. Oud et al. (2000) tested and found related acoustic respiratory sound analysis and pulmonary disorder. The results showed that disease conditions could be recognized by an acoustic footprint. However, the method required quiet laboratory conditions, laboratory technicians and signal processing, discouraging its use as a simple patient-accessible test. The repeatability of the measures depends upon the patient exerting the same amount of effort at each test instance.

Spirometric Assessment

Spirometry assesses respiratory characteristics by measuring respiratory flow rates during a patient's maximal exhalation effort. Currently, spirometry dominates respiratory assessment, especially in obstructive disease (American Thoracic Society, 1999; Snow, 2000). Portable spirometric devices are considered an adequate solution for home monitoring (Eid et al., 2000). Stein et al. (1966) have demonstrated correlation between spirometric measurements and RAW thus validating spirometry's application in obstructive pulmonary disease home and clinical monitoring. Since spirometry is prevalent among respiratory measurement techniques, it is worth examining its details to examine those measurements that are the benchmark for clinics. The qualities that have made spirometric devices successful in the market are worth examining as a model for APD development.

Spirometry requires a patient to forcefully expel air from the respiratory system with maximal effort. Since the amount of air expelled over time depends on lung-volume changes and resistance to airflow, the technique can diagnose COPD, asthma and other respiratory diseases that cause airway constriction and respiratory tissue change.

The results of the spirometric forced exhalation can be divided into at least three diagnostic components. The first, peak expiratory flow (PEF), is the air volume exhaled per time measured in the first 150 milliseconds of the expiration. The PEF is strongly influenced by the thoracic and abdominal muscles, and represents expiration mainly from the large upper respiratory bronchial pathways (Eid et al., 2000). The second

component, forced expiratory volume in one second (FEV1), reflects large and medium sized airways at lower lung volumes (Eid et al., 2000). The third component, forced expiratory flow from 25% to 75% of vital capacity ($FEV_{25\text{-}75\%}$) reflects the function of small airways (Eid et al., 2000). FEV1 and $FEV_{25\text{-}75\%}$ depend less on effort of the exhalation and more upon the elastic recoil of the lungs because they occur later in exhalation procedure.

The American Thoracic Society has published criteria to standardize pulmonary function test (PFT) equipment performance (1995). The criteria specify maximum allowable inaccuracy, imprecision and total error for forced vital capacity (FVC), FEV1, $FEV_{25\text{-}75\%}$, and PEF, in both ambient and body temperature, water vapor saturated (BTPS) conditions. The guidelines require testing with a standard spirometry simulator that generates standardized waveforms and require a device resistance less than 1.5 cmH20/L/sec from 0 to 14 L/sec airflow. QRS Diagnostics, LLC. (see Table 1) used the guidelines to demonstrate the performance of its product, and verified that their product, Sensaire, met the human subject criterion by comparing its results with that of a standardized spirometer for two subjects (Crapo and Jensen, 2001).

The most abundantly available device for home use without technician assistance is the peak flow meter (PFM). For examples, see the MicroMedical turbine meter, the Ferraris fdE pocket meter, the Vitalograph meter, the MultiSPIRO meter and mini-Wright meter in Table 1. The popularity of PFMs can probably be attributed to their inexpensive, simple design based on Wright and McKerrow's (1959) original or some slight variant. These are designed to measure from zero to up to 1000 L/min

instantaneous peak flow in some models. The simple mechanical design keeps prices low. There are slightly more expensive digital models available as well.

Since spirometry, and in particular PEF is extremely effort dependent, it is recommended that the PEF measurement is repeated until 10% reproducibility is achieved and that patient PEF measurement methods be carefully assessed (Janson-Bjerklie and Snell, 1988; Jones and Mullee, 1990). PFM users are directed to find their maximum achievable PFM reading and then compare daily readings with this maximum to determine their current lung function as a fraction of their maximum PFM reading. A patient's PEF value is typically compared with the personal best by placing it into one of the three categories: green 80 to 100 percent of personal best, yellow: 50 to 80 percent of personal best (asthma conditions may be deteriorating), red: less than 50 percent of personal best (take medication and consult physician). Mechanical PFM modules typically have markers on the body that indicate maximum PFM and the moderate and danger ranges of respiratory impairment. Digital modules implement this through electronics and can store several hundred readings. Some PFM units are capable of transmitting data back to a clinic (see Stahlman and Salmun, 1999).

Table 1. Selected commercially available peak flow meter (PFM) and pulmonary function test (PFT) device models

Distributor	PFM Manufacturer/ Model	(M)/ (E)	Range (L/ min)	Price
Allergy Control Products,	Personal Best	(M)	60- 810	$26
Inc.	Personal Best LFM	(M)	50- 390	$26
96 Danbury Road	Assess	(M)	60- 880	$19
Ridgefield, CT 06877	Assess LFM	(M)	10- 380	$21
800.422.DUST				
Asthma Stuff	Pocket Peak	(M)		$22
P.O. Box 1537	Clement Clarke:			
Amherst, New York 14226	- Airzone	(M)		$19
866.943.3937	- Mini-Wright Std	(M)	60- 800	$29
	- Mini-Wright LFM	(M)	30- 400	$29
Pulmonary Data Services,	KoKo PFM	(E)	60- 840	$40
Inc.	KoKo Peak Pro PFM	(E)	60- 840	$75
Ferraris Medical, Inc.	KoKo Software Home	(E)		$60
908 Main Street	KoKo Software Pro	(E)		$250
Louisville, CO 80027				
800.574.7374				
QRS Diagnostic, LLC.	SpiroCard PC Card	(E)		$1,495
P.O. Box 47304	PFT	(E)		$1,995
Plymouth, MN 55447	SpirOxCard PC Card	(E)		$1,995
800.465.8408	PFT/O2			
	Sensaire Portable PFT			
	Device			
Abbreviations: LFM = low flow model, (M) = mechanical, (E) = electronic				

Supplementing PEF measurements with FEV1 and $FEV_{25\text{-}75\%}$ to obtain more accurate and complete respiratory data presents difficulties for home measurement, but these capabilities have been incorporated into some models. These measures require more complex and larger devices to record larger volume and time measurements. The analysis to calculate the values is more complex and the PFT devices supporting these functions are more expensive though some devices, such as the KoKo Peak Pro offer peak flow data as well as some PFT data while maintaining a low cost (see Table 1).

Complete spirometry requires technician guidance for coaching the patient throughout the maneuver. Klaustermeyer et al. recommend that PEF be used only in conjunction with full laboratory spirometry tests (1990). Further, as FEV1 decreases, the sensation of effort increases (American Thoracic Society, 1998), hence introducing the possibility of cyclic reduction in FEV1 for a single patient, and underestimating FEV1, especially in the absence of a trained technician. If such measurements are unavailable at home, it may be necessary to augment home respiratory health records with recurrent clinic visits.

Though spirometry and PFMs dominate respiratory function testing, several complications arise regarding the data they provide. Spirometry is criticized for its effort dependence, a quality that casts doubt upon the measurement's repeatability and representation of the patient's respiratory system under its normal breathing state. Some studies as previously mentioned have shown however that with adequate coaching, results are repeatable and truly indicate airway obstruction and restriction (American Thoracic Society, 1999). Unfortunately, the same coaching that helps to produce a repeatable measurement is often unpalatable to younger children (Malmberg et al., 2000). Further, consistent coaching could only occur in a clinical setting. In clinical and emergency settings, the portability and low cost of spirometry is attractive, but its effort-dependent quality precludes its use on paralyzed, unconscious or ventilated patients.

In home monitoring programs, often focused on prevalent reversible obstructive diseases, spirometry falls short of a being complete predictive tool. PFMs and

spirometry lack the capabilities and sensitivity to directly measure short-term small RR changes. Snow et al. (1995) explain that FEV1 and PEF measurements may fail to show important changes in RAW because patients can often exceed the pressures required for maximal expiration. Bronchoconstriction can occur before asthma symptoms, and thus may go undetected when spirometry is used for detection. Eid et al. (2000) found that PEF values in the normal range were least likely to predict whether or not a child would develop or presently had moderate to severe asthma. Hence, the subject group most at risk for asthma attacks was most poorly identified by PEF. Gern et al. (1997), Fleming et al. (1999) and Rakes et al. (1999) found that allergic reaction to Rhinovirus resulted in inflamed lower and smaller airways. This finding renders PFM measurements less indicatory of the true state of respiratory distress because PFM measurements were shown to be dominated by upper airway characteristics. Thus in many cases spirometry and especially PFMs lack the capability for sensitive preventative diagnostics. Resistance is a more direct measure of airways constriction whether due to inflammation, asthmatic bronchospasmic restriction or other obstruction. Isolating resistance and then comparing with forced flow may augment respiratory monitoring in a home setting.

Esophageal Balloon

Direct resistance measurements that lack the need for specific patient breathing patterns eliminate the effort-dependence and affective component of forced exhalation studies. Esophageal balloon techniques allow direct measurement of pressures across

the respiratory system. The esophageal balloon technique makes possible continuous flow and pressure signal analysis. However, the Mead and Whittenberger (1953) and Frank et al. (1957) esophageal balloon analysis technique requires the analysis of full breaths that start and return to the same lung volume. Thus the technique has limited temporal resolution – that of the frequency of occurrence of good full breaths - and cannot be used to distinguish inhalation from exhalation resistances, during which different physiological forces act upon the airways. Further, this pressure and flow sensing technique for RR analysis requires technicians and invades the patient, prohibiting widespread and home use.

Full Body Plethysmography

The full body plethysmograph (FBP) technique offers advantages in respiratory characteristic measurement but its requirements limits the extent of its use. FBP places the subject in a sealed enclosure primarily to measure airways conductance by monitoring pressure and flow. The technique's strength lies in the facts that its primary measurement is airway resistance and that it can be used to measure thoracic gas volume. Snow (1997) emphasizes the importance of measuring airway resistance and thoracic gas volume to detect the condition when a subject compensates for airway obstruction by increasing functional residual capacity. This case has the effect of reducing the effect of airway constriction by expanding the lungs to a state of larger airway caliber. FBP focuses on the state of the airways, rather than total RR, making more specific its measurement information in the diagnosis of airways problems.

Lausted (1997) presents a review of FBP technique relevant to APD development and describes the manner in which APD RR correlates with FBP RAW. Previously, FBP's requirement for patient cooperation in performing special panting maneuvers has been cited as a weakness in plethysmography application. However, Krell et al. (1984) found that quiet breathing FBP produced measurements comparable to those achieved by panting maneuvers. Despite the value of its measurements, FBP's primary disadvantages in widespread home and clinical application are the size and cost of the apparatus and the need for a technician to instruct the subject and perform the measurements.

Interrupter Devices

Airflow interruption devices have a potential for marketability as they are non-invasive and can be made portable. Like a shutter, the interrupter occludes airflow for an instant and then computes the RR from the change in pressure during the occlusion and the flow just prior. The interrupter technique can provide fairly continuous RR measurements, but suffers inaccuracies from inertial effects and is dependent on respiratory compliance (Kessler et al., 1999). Bates et al. (1988) found that accurate interrupter measurements depended heavily on proper technique and were confounded by any compliance inequalities portions of subjects' lungs (1988). These qualities could be considered to limit interrupter use to laboratories or clinics.

Forced Oscillation Technique: Pressure Addition

Forced oscillation (FO) is a technique that non-invasively measures RR and has potential for widespread use. In obstructive disease assessment FO is more suitable than spirometry for children because FO is non-invasive and effort independent. FO uses a loudspeaker to add a pressure signal, usually between 2 and 10 Hz to a patient's breathing pattern and measures pressure and flow variation during the oscillations to measure RR. Lausted has reviewed aspects of FO that relate to APD development, particularly highlighting the special breathing patterns and supervision FO required in laboratory settings (1997).

In its early development, FO had been restricted to laboratory use, but new developments have shown that FO can be implemented in smaller units that operate in a manner suitable for any user. Recent work has shown that FO can provide accurate measurements during passive breathing. Tests have shown FO to provide accurate measurements in children and at home without supervision (Ducharme and Davis, 1997). These passive breathing FO measurement techniques have proven useful in paralyzed and ventilated patients, requiring only the minor modification of ventilation apparatus (Navajas and Farre, 2001). The passive breathing FO technique has been used to monitor sleep breathing disorders (Lemes and Melo 2003).

The major limitation to FO use would be its required components. A recent description of an FO system included a PC, DAQ and 6-inch speaker (Melo and Lemes, 2002). These requirements would preclude its use as an inexpensive or portable and compact device. The PC and DAQ functions could be integrated into a small unit, but

FO would still require the speaker and speaker driving power that might make impractical a hand held FO configuration.

Perturbation Techniques and the APD: Flow Reduction

Perturbation techniques briefly, partially occlude airflow several times each second during passive breathing. In the sense that it reduces airflow, perturbation is similar to the interrupter technique and in the sense that it imposes an additional signal on the breathing waveform, it is similar to the FO method. The APD (Johnson et al. 1984a, Johnson et al. 1984b), measures RR by calculating the ratio of pressure change to flow change during the airflow perturbation. The current configuration of the APD requires a data acquisition card and computer for analysis, largely confining its use to the laboratory.

The mechanisms by which the APD functions have been documented extensively (Lausted 1997) and hence have been excluded from this discussion. Its value lies in its lack of forced breathing requirements, its non-invasive measurement technique, its sensitivity and capability for fine time-scale measurements. Patients breathe normally into the device and hence have a reproducible baseline for measurements. In contrast to FBP and esophageal balloon techniques but like interrupter and FO methods, the device measures total RR at each calculation, as evident in the response of chest wall accelerometers (Lausted, 1997).

As discussed prior, it is useful to consider whether the APD alters the respiratory system from its natural state in order to make a measurement. The start of any such

undesired effect might initially be seen through EEG potentials, indicating an imminent respiratory response that would occur in response to perturbing the respiratory system. Akay and Daubenspeck (2000) measured brain EEG potentials induced 200-ms pulses at negative pressures from 6 to 17-cmH20 at the mouth and found delays from 50 to 100 ms in the EEG response to the pulse. APD perturbations last about 100 ms for a 10 Hz perturbation frequency with pressure magnitudes around +/- 3.0 cmH20 (1.2 inH20). Since the pressures of perturbation are small and short in relation to those used in the study, their impact on the human respiratory control system might be small. However, this might be examined further in the future.

The APD's potential for quick response without tiring the user through repeated forced expiration opens the door to other applications. The APD presents potential for biofeedback studies in bronchodilation. Biofeedback is the ability to change aspects of the body's functioning that are typically under autonomous control. For example, patients have learned to control the form of their brainwaves on EEG displays or muscle activity visible on EMGs (Olton, 1980). An asthmatic patient might learn to control that aspect of autonomous function that controls airways constriction. The APD's fast response time also facilitates the characterization of a respiratory system throughout a range of positions within the breathing cycle.

OBJECTIVES

To meet the aforementioned medical industry need identified in the Literature Review, the APD requires modification. The following design objectives define the manner in which modification would allow further APD implementation in the health care industry.

1. To revise the APD design such that a single stand-alone unit performs
    a. data collection;
    b. data analysis;
    c. a display;
    d. IO circuitry for expanded functionality.
2. To design a framework for a fast-response APD.
3. To create a computer interface with the APD for expanded data transfer and calibration protocols.

Design Specifications

The following design specifications detail the characteristics that the APD-SA must possess in order to meet the objectives while providing measurements comparable to the current APD design:

1) Prior design specifications that should be preserved

   a. Variable direct current voltage (VDC) motor: 2W or less, 250 mA maximum continuous current or less (*e.g.* A-max16 12V 016 EBCLL 2W SL 1SH, Maxon Precision Motors, Inc., 838 Mitten Rd., Burlingame, CA 94010)

   b. Rotating, belt-driven perturbation wheel perturbing airflow near 6 Hz

   c. Accommodations for Fleisch pneumotachograph (PT)

      i. Pressure sensing for pressure differential across the PT: +/- 1 inH20 (maximum pressure gradient across the PT is 0.4 inH20)

      ii. Mouth pressure sensing +/- 10

      iii. Mechanical provision for securing PT to the body

      iv. Electrical provision for 6 Ohm PT heating element: 6VAC @ 1 A

   d. Analog to digital converter (ADC): 12-Bit or greater at 500 Hz

   e. Device resistance adjustment

   f. Software

      i. Digital filter after analog stage

      ii. Perturbation detection and RR measurement based on perturbation magnitude – developed as a modification to the prior method (Lausted, 1997, Lausted and Johnson, 1998)

2) New design specifications

   a. Stand-alone system operation

      i. Character display and supporting circuitry

      ii. Simple user interface

      iii. Measurement mode is the default power-on mode

iv. Calibration mode and calibration variable retention in the device

b. Hardware requirements

i. Single system voltage supply input for potential battery operation without heated PT and design simplicity

ii. On-board ADC

iii. Sufficient random access memory (RAM) to store pressure, flow and RR value data points required for the proper display of a value:

1. Preliminary testing: 50-data point buffer for pressure, flow, device resistance, derivative of device resistance and RR values: 250 x 32-bits per value = 1 kilobytes minimum data RAM
2. Preliminary testing: intermediate variable storage and constants: 240 bytes minimum RAM allows about 60 x 32-bit variables

iv. Sufficient program storage space

1. Preliminary testing: 100 kilobytes, including compiled IO routine support.

v. central processing unit (CPU)-controlled pulse-width modulated (PWM) 40 kHz typical sourcing up to 200 mA or with external drivers for additional current

vi. Circuit layout

1. separation of analog and digital grounds
2. separation of analog and digital circuit components

3. separately regulated analog and digital supplies

vii. IO Support

1. To transmit raw pressure and flow data: 2 Channels ADC @ 500 Hz, IEEE 32-bit gives 32 kilobits per seconds
2. To transmit each RR value only: 10 Hz nominal at IEEE 32-bit precision, transmit time, press, flow at each RR value = 4 kilobits per second

c. Firmware

i. Tune software execution speed to ensure proper ADC control

ii. Effective digital filtering and spike rejection

iii. Software circular buffering for all data and RR values

iv. RR value tracking for display of averaged RR

v. RR value update at each new valid perturbation for maximum real-time RR value display, if fast-response is used

vi. Respond to IO requests

3) A hardware platform for the stand-alone APD-SA must have the following characteristics

a. Availability: Either:

i. fabricable within the current ENBE departmental means

ii. or affordable fabrication outside the department

iii. or easily acquired from vendors with assured product longevity

b. Data Rates: Capable of the necessary data rates including

i. Adequate data input/ output (IO) rates

ii. Adequate data analysis speed (*i.e.* sufficiently fast processor instruction execution); initial profiling found a minimum requirement of 100MHz Pentium in Windows 98 for real-time processing that does not require the addition of a peripheral memory chip design

c. Memory and Storage: Contain adequate memory, as specified in section 2) for:

i. Program storage

ii. Temporary data storage

4) Host PC Software

a. Support for the APD-SA should be integrated into the existing APD100 host software

b. Provide any calibration routines not supported by the stand-alone device

c. Provide pressure, flow and RR values from the APD-SA while connected

d. Provide data viewing and export functions for the APD-SA

e. Indicate what mode the software is using, *i.e.* APD-SA or APD100 with a DAQ

EQUIPMENT

APD Design Prior to this Research: APD100

As described in detail in Lausted (1997), APD100 consists of the APD body, power supply unit and computer data acquisition and analysis system. The APD body conducts air flow through the PT, the variable device resistance tube and to the rotating wheel that intermittently reduces airflow. The APD body also houses the components that create 1) the flow signal: the proximal and distal PT pressure taps, differential pressure sensor and filter circuitry and 2) the pressure signal: the proximal PT pressure tap, differential pressure sensor and filter circuitry. A male D-Subminiature 9-pin (DB9-M) transfers to and from external devices the pressure and flow signals as well as power for the sensors, perturbation wheel motor and PT heater. The power supply unit transforms 115 VAC to 6.3 VAC PT heater supply voltage and variable VDC motor supply voltage.

A computer data acquisition and analysis system contains the DAQ and analysis software. The DAQ supplies 5 VDC to the sensors and converts to digital format the analog pressure and flow signals. The APD100 software, written in Microsoft Visual Basic, analyzes the data to determine respiratory resistance. Device drivers provided by the DAQ manufacturers control the DAQ and organize the converted digital data into buffers. APD100 software communicates to the drivers the buffer size, channels on

which to collect data and sample frequency. The sample frequency is 500 Hz and the buffers contain five seconds of data from four channels, the first two, channels zero and one, logging flow and pressure data respectively. APD100 offers data logging and viewing options, and contains device calibration routines.

APD100 data collection and processing uses a dual, alternating five-second, 500 Hz data buffer scheme. When the DAQ finishes filling one buffer, the software is notified that the filled buffer is available for processing. The DAQ continues sampling, filling a second buffer while the first buffer is processed. APD100 looks through each filled buffer, determines likely perturbations, determines perturbation frequency and affirms or denies each perturbation's integrity. Each approved perturbation is used for a RR measurement. The processing loop continues until the number of inhalation and exhalation perturbation RR measurements exceed the maximum specified number, typically 100.

Equipment Available: Options for APD Design Revisions

The next step in APD development could have taken many forms and still met the research objectives and design specifications. Appendix H includes information on hardware and software technology combinations that might have been used in this research, as well as a comparison of the relevant qualities of each. A form was selected for the APD-SA USB that was seen to be the best compromise in design possibilities to achieve:

- low development cost
- feasible prototype fabrication given the available time and resources
- integration with existing APD technology

Equipment: Revised Design for APD-SA USB

Figure 1 illustrates the major components in the revised APD-SA USB design. An electronics enclosure is mounted on the original APD body. The APD body modulates breathing airflow and generates pressure and flow signals. The electronics enclosure analyzes pressure and flow signals to determine RR, controls the APD body motor, controls the LCD display, and coordinates USB communication.

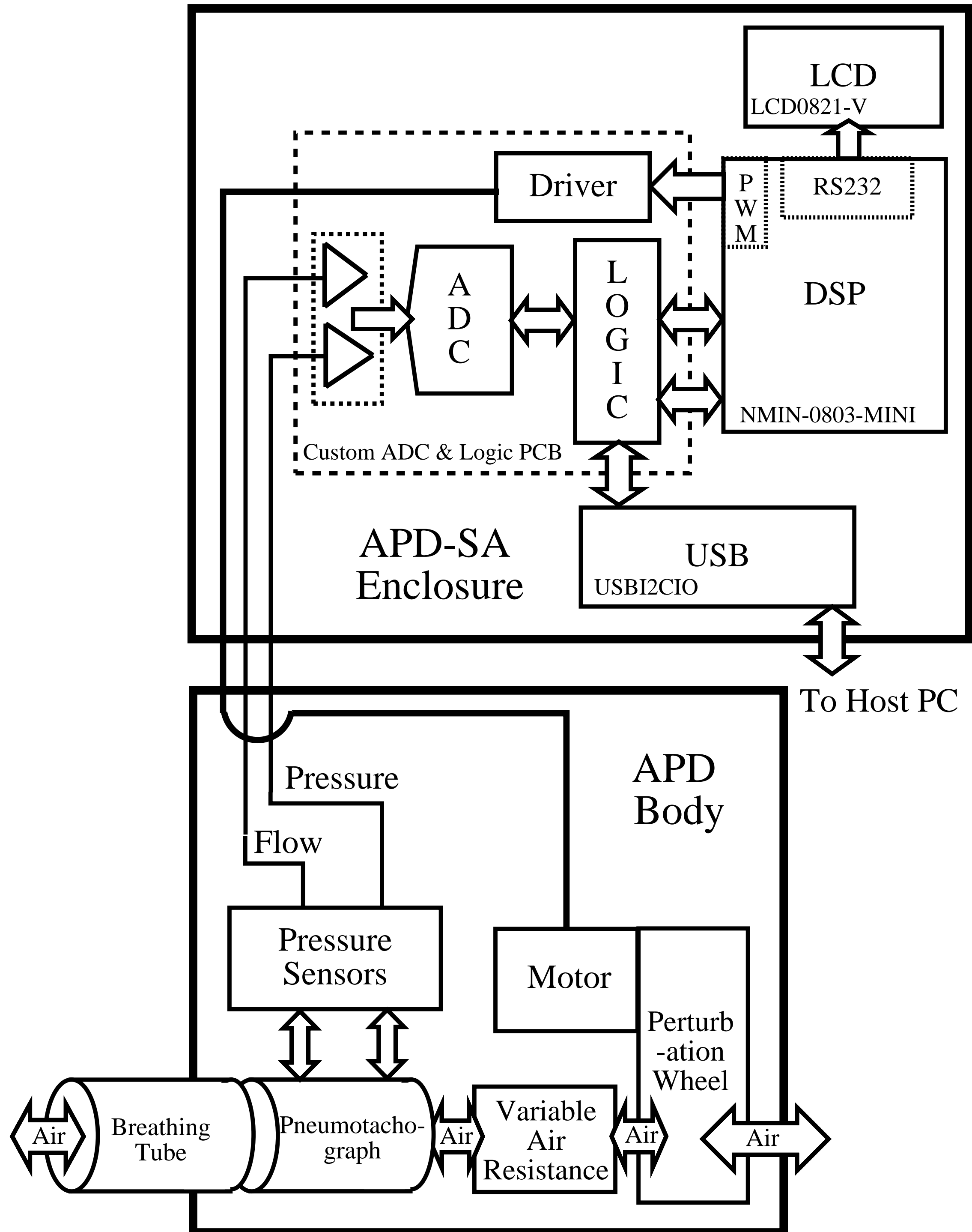


Figure 1. Block diagram of the design for the APD-SA USB

APD-SA Components

The APD-SA USB system comprises the APD body, electronics enclosure, PC-hosted APD200 software for USB-based communication, a DC power adapter and USB cable. The APD body is the same body used in the APD100 system; it contains the same PT, sensors, and perturbation mechanisms described earlier. Since the APD-SA was designed to be integrated with existing portions of APD100, the electronics enclosure mounts directly atop the APD body and the APD200 software is an upgrade to the APD100 software. APD200 integrates USB communication and APD-SA-specific routines into the APD program and was developed from Microsoft Visual Basic 6.0 on a Windows 98 PC. Figure 2 is a rendering from CAD drawings that shows the rectangular APD-SA enclosure mounted to the APD body via mounting brackets, one on each side of the device. A liquid crystal display (LCD) screen faces the user to display device status, measurement progress and an average RR value when enough usable perturbations have been collected. A DB9-F on the underside of the APD-SA enclosure connects to the DB9-M connector on the APD body to transfer signals between the two. DC power and USB connections are on the lower rear face of the enclosure. Through APD-SA electronics, power is distributed to the motor, sensors and processing circuitry.

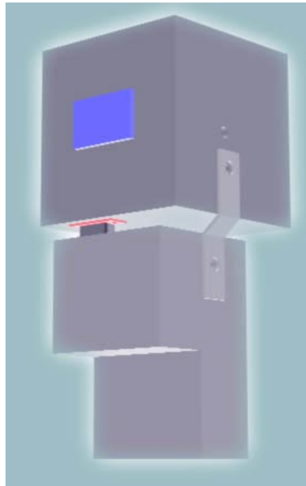


Figure 2. A rendering of the APD-SA in which the APD-SA enclosure housing electronics mounts onto the APD body that is also used in the APD100 system

The APD-SA enclosure houses all electronics required to measure RR. Electronic components are located on one of four printed circuit boards (PCBs) summarized in Table 2. A custom ADC and logic PCB serves to interface all electronics by providing connection points for cables, logic to translate and control integrated circuit (IC) inter-communication and power distribution from the single DC power input to the APD-SA enclosure. Figure 3 is a schematic of the APD-SA custom ADC and logic PCBs. The images for the PCB layout can be found in Appendix B. Figure 4 shows how the four PCBs and their associated ICs are interconnected to transfer data.

Table 2. Summary information for the four printed circuit boards in the APD-SA enclosure

Function & Description	Manufacturer Information	Part No.
ADC & Logic – analog signal conversion, communication interface logic and power distribution	Custom	N/A
LCD – RS232/ I2C backlit display	Matrix Orbital Corp., Calgary, Alberta, Canada	LCD0821-V
DSP – collect, analyze and communicate data	New Micros, Inc., Dallas, Texas	NMIN-0803-MINI
USB – control USB communication with PC host	DeVaSys, Penfield, New York	USBI2CIO

Each PCB required additional software or hardware tools either for design or development:

- Custom ADC and logic PCB
    - The PCB layout was designed in a schematics and layout software package called Eagle 4.11 (CadSoft Computer Inc., Delray Beach, Florida).
    - The layout was printed on a LaserJet 4L printer (Hewlett Packard, Palo Alto, California) on laser printer transparency film CG3300 (3M, St. Paul, Minnesota)
    - PCB was etched on single-sided copper clad pre-sensitized positive etch board using an exposure kit 416-X and etching photofabrication kit 416-K (MG Chemicals, Surrey, British Columbia, Canada)
    - PCB holes were drilled using DB-0360 #64 or 0.0360-in diameter 1/8-in shank drills (T-Tech, Inc., Norcross, Georgia)
- LCD PCB
    - A custom null-modem cable was created to interface the 0.100-in serial communication (RS232) headers on the LCD platform to a DB9 PC serial port

- Digital Signal Processor (DSP) PCB
    - A Joint Test Action Group (JTAG) FLASH programming and DSP debugging cable was obtained (Gregor Air, Inc., San Jose, California)
    - DSP firmware was developed in C and assembly language using CodeWarrior integrated development environment (IDE) Embedded for Motorola DSP56800 Academic Edition (Metrowerks, Austin, Texas)
- EZUSB PCB
    - EZUSB firmware was developed in C using uVision IDE for 8051 (Keil Software, Inc., Plano, Texas)
    - Microsoft Visual Basic 6.0 was used to develop host PC USB application code

Additional tools and components were used to interface components and development platforms, but are generally part of a standard electronics development laboratory. This includes items such as test and measurements tools, cables, connectors, hand tools and breadboards.

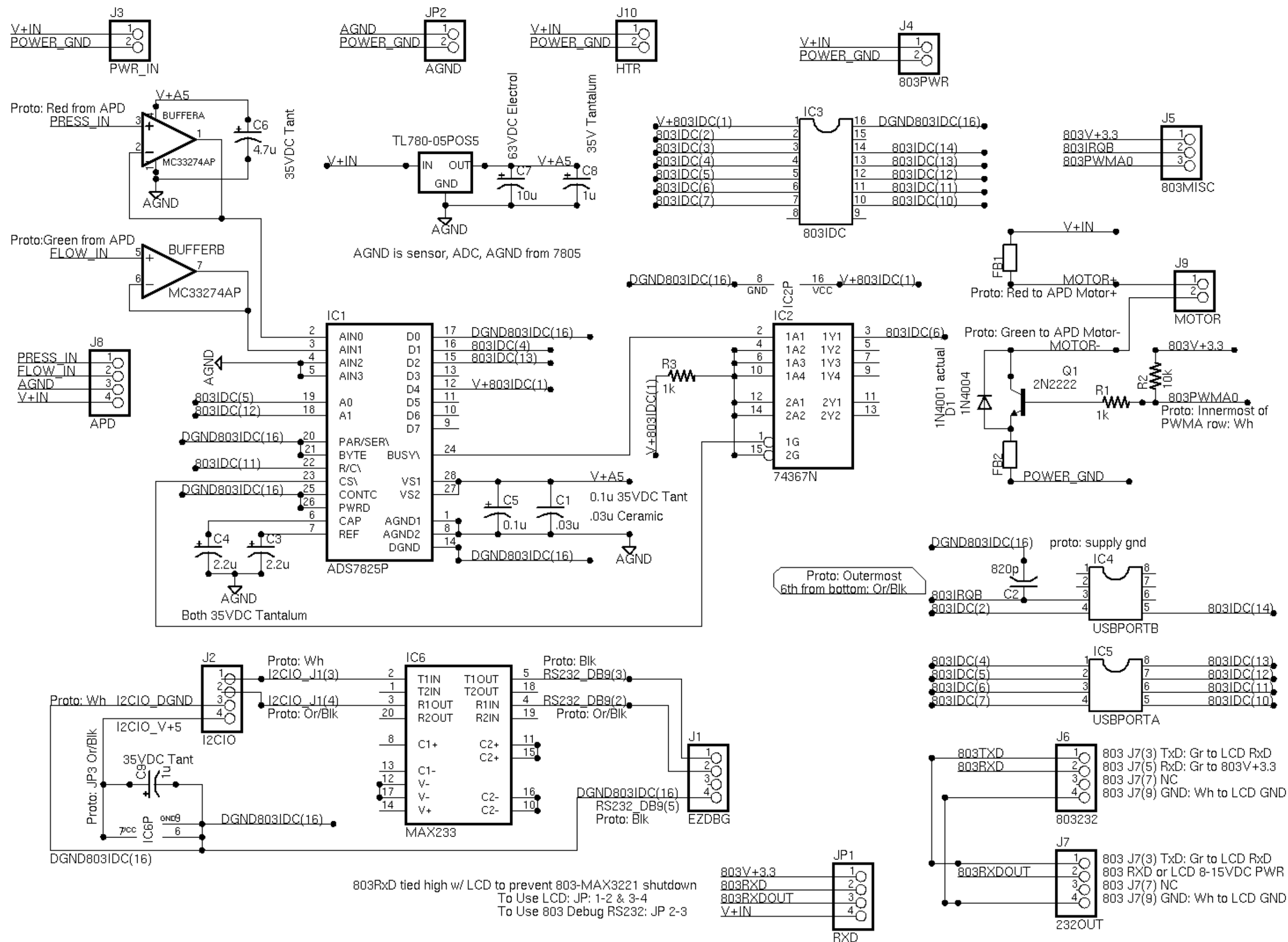


Figure 3. APD-SA ADC and logic schematic for analog signal conversion, motor control and interfacing APD-SA PCBs; the figure includes all headers for connections between PCBs; all electrical connections, or nets, are named, and identical names indicate electrical connection, even if a trace is not drawn between the named points; the PCB layout was derived from this complete schematic

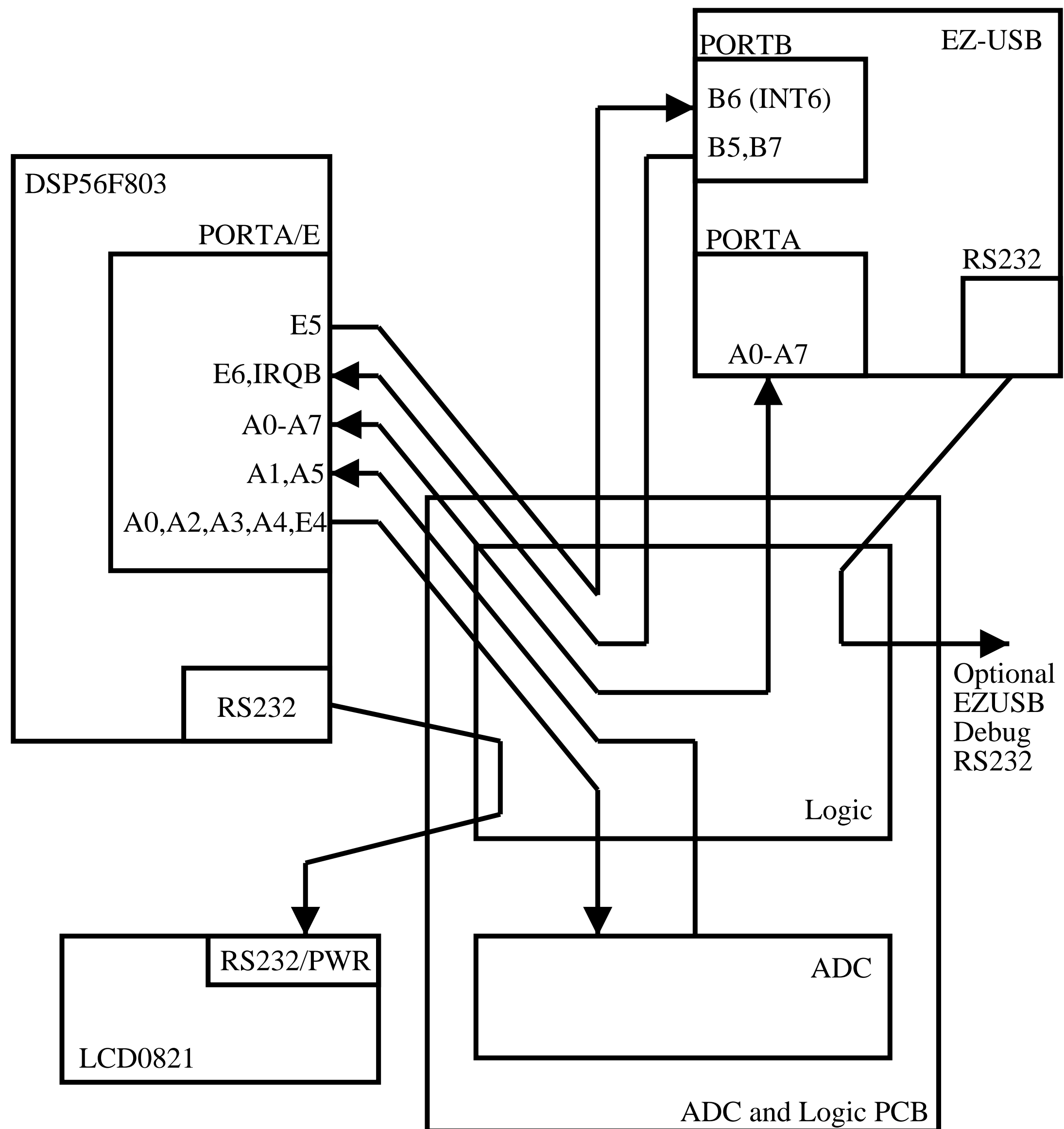


Figure 4. Interconnections and data exchange among the ICs and PCBs in the APD-SA enclosure

Appendix A contains detailed CAD drawings that describe the enclosure dimensions, electronics mounting points in the enclosure, orientation of the PCBs and assembly procedure. The mounting positions for the PCBs were determined by the physical size of the PCBs as well as the clearance required for connectors. The enclosure was designed so that once the enclosure is assembled, the sides of the

enclosure can be removed with a single bolt to expose the DSP and USB development platform connectors that allow firmware updates and debugging. There is clearance between components to attach the JTAG connector to the DSP platform and clearance to connect a cable to an RS232 debug port for the EZUSB platform.

APD-SA Operation

Division of Functionality

Functions performed by the APD-SA USB system are divided into those performed by the stand-alone device and those performed by the APD200 software and host computer. The APD body portion of the stand-alone device generates the pressure and flow signals that perturb the breath. The APD-SA enclosure portion of the stand-alone device controls the motor speed, and digitizes and collects the pressure and flow signals to calculate RR from well-proportioned or good perturbations – those with large enough signal-to-noise ratios.

When power is applied to the APD-SA, the unit goes through its initialization steps and then continually reads data and calculates RR until the specified maximum number of perturbations is attained or the EZUSB platform asserts itself to the DSP platform. The EZUSB platform is powered by the USB bus power from the host PC and thus does not function until a USB cable is connected from host PC to APD-SA. Figure 5 illustrates how the DSP, EZUSB and host PC signal each other through interrupt requests to orchestrate the data transfer that underlies the APD-SA operation.

Host PC to APD-SA USB transfer requests are orchestrated through USB endpoint interrupts. Transfer from the DSP to the EZUSB is signaled by the EZUSB external hardware interrupt six and the EZUSB signals the DSP through its external hardware interrupt B.

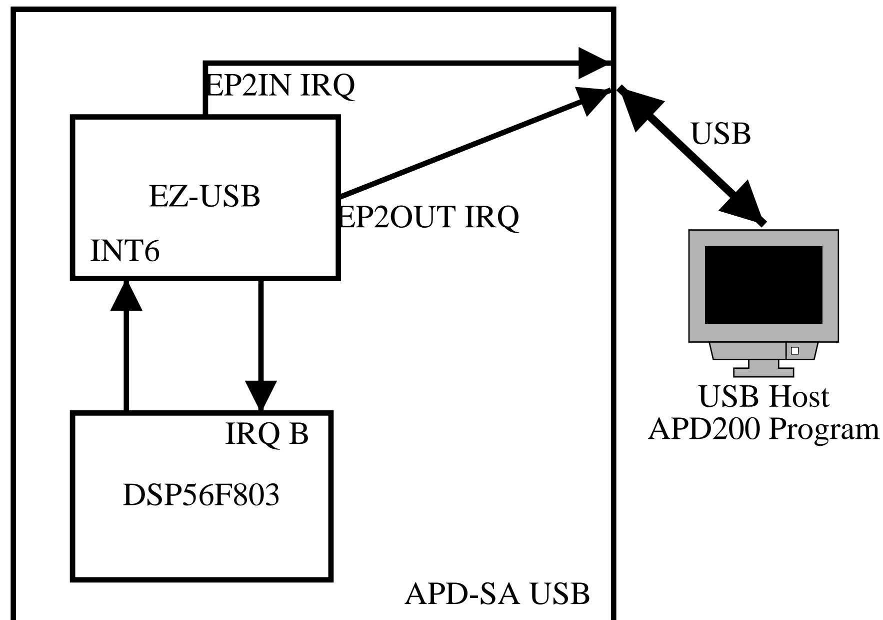


Figure 5. Interrupt driven communication among APD-SA DSP, USB and host PC

APD200 handles APD-SA USB calibration and detailed RR data collection during measurement. APD200 can initiate raw data streams, RR measurement data streams and calibration variable transfers. The streamed data is used for calibration routines. Calibration variables are stored in the DSP FLASH memory. APD200 includes routines for plotting and saving the detailed RR measurement information. Calibration information for the APD-SA USB can also be stored in files on the host PC.

Firmware Descriptions

USB firmware was written to cause the APD-SA to appear to the host computer as a human interface device (HID) and transfer commands and data between the host PC and DSP. USB firmware was developed on the same PCB platform used in the APD-SA enclosure, using the firmware and software development tools listed earlier. USB and RS232 null modem cables allowed debugging. Figure 6 summarizes the USB code. The firmware loops until it receives a request from the USB host. The request arrives as a block of data, the first byte of which contains custom-defined codes that indicate the type of operation to be performed. The EZUSB then signals the DSP with the appropriate corresponding command, awaits acknowledgment, passes acknowledgment to the host and transfers data when ready until the operation is complete.

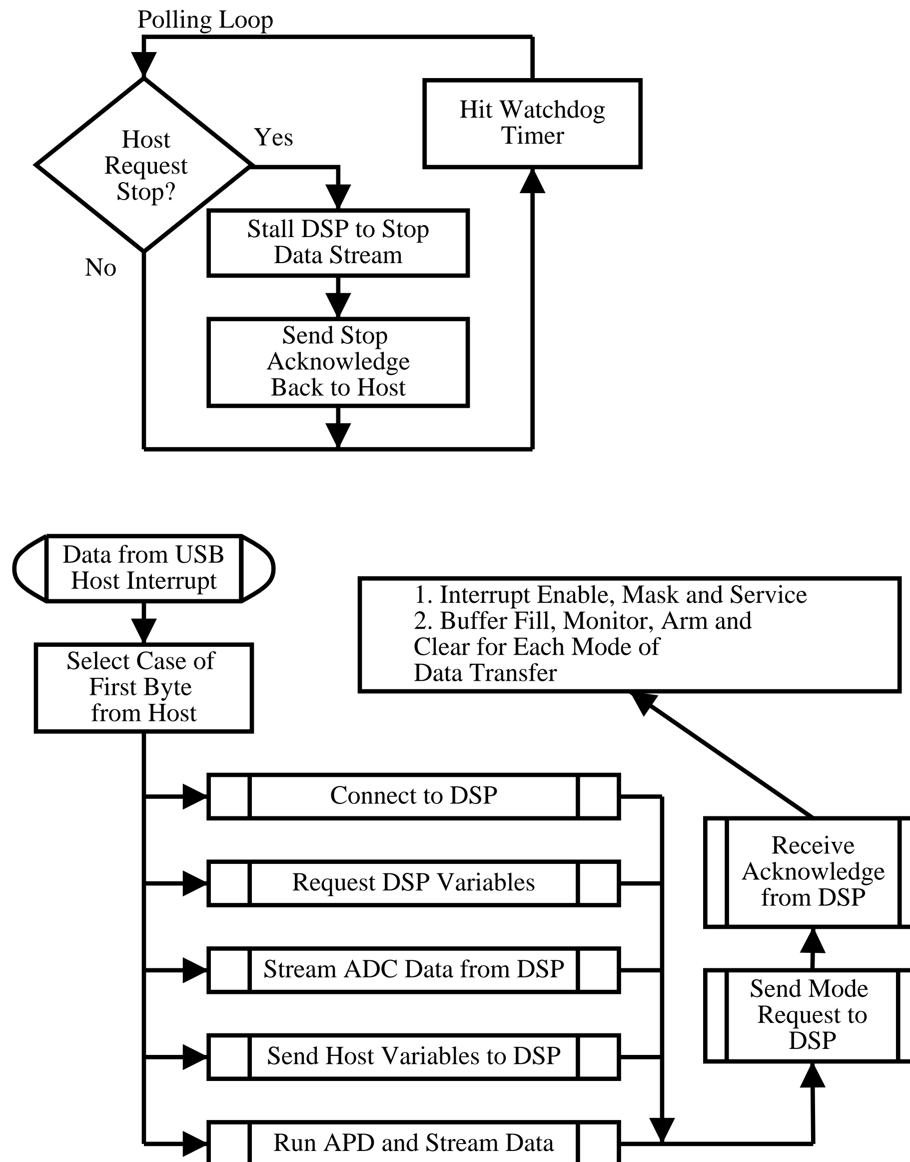


Figure 6. Diagram of the APD-SA EZUSB firmware; the polling loop (top) executes continuously; if a command is received from the USB host PC, then the EZUSB executes the necessary actions to comply with the command (bottom)

DSP firmware was written to coordinate ADC control, data buffering, data processing, motor speed, LCD updates and responses to EZUSB requests. When power is first applied to the APD-SA, the DSP initializes its circuitry, measures baseline (no pressure and no flow) sensor voltages and then continually reads data, looking for

perturbations. The unit displays to the user through the LCD its progress through these initialization steps. When good perturbations are detected, the LCD screen indicates progress toward the maximum number of perturbations to collect before showing average RR. When the maximum is reached, the DSP stops measurements and displays the average of inhalation and exhalation RR. As depicted in Figure 7, at any point during RR measurement, if the EZUSB asserts itself to the DSP, the DSP will immediately go into a USB slave operation mode in which it only acts after receiving a command passed to it via EZUSB from the PC host.

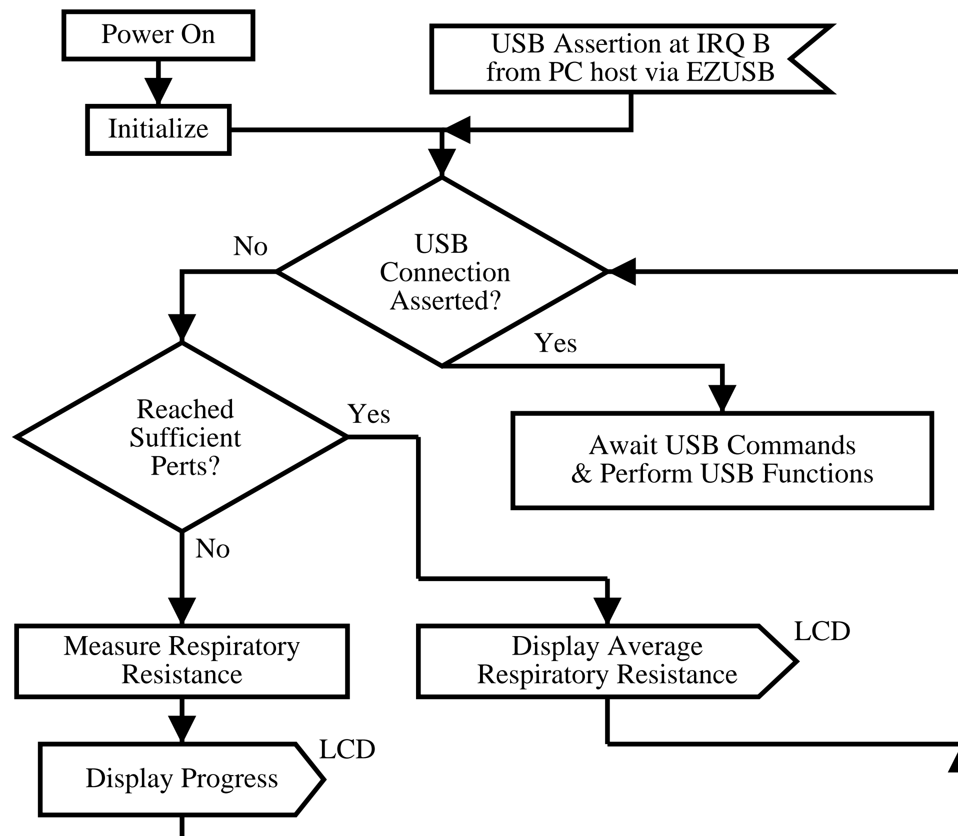


Figure 7. Diagram of APD-SA DSP firmware operations upon power-up and the point at which the DSP will heed PC-initiated USB commands

To achieve the stand-alone and USB-controlled functions, a polling loop in the DSP firmware toggles among operating modes. Once a mode is selected, a DSP timer

is used to periodically execute functions. Figure 8 illustrates how the loop toggles among modes after power-up and how a DSP timer triggers software interrupts that periodically call procedures. Timer interrupt-called procedures are prefixed as “TimeProc,” as in “TimeProcRunMode” and “TimeProcStreamMode.” Depending on operation mode, collected and analyzed data will be transmitted to the EZUSB, as needed. The “TimeProcRunMode” timer interrupt function is the RR measuring function. Figure 9 illustrates its major components.

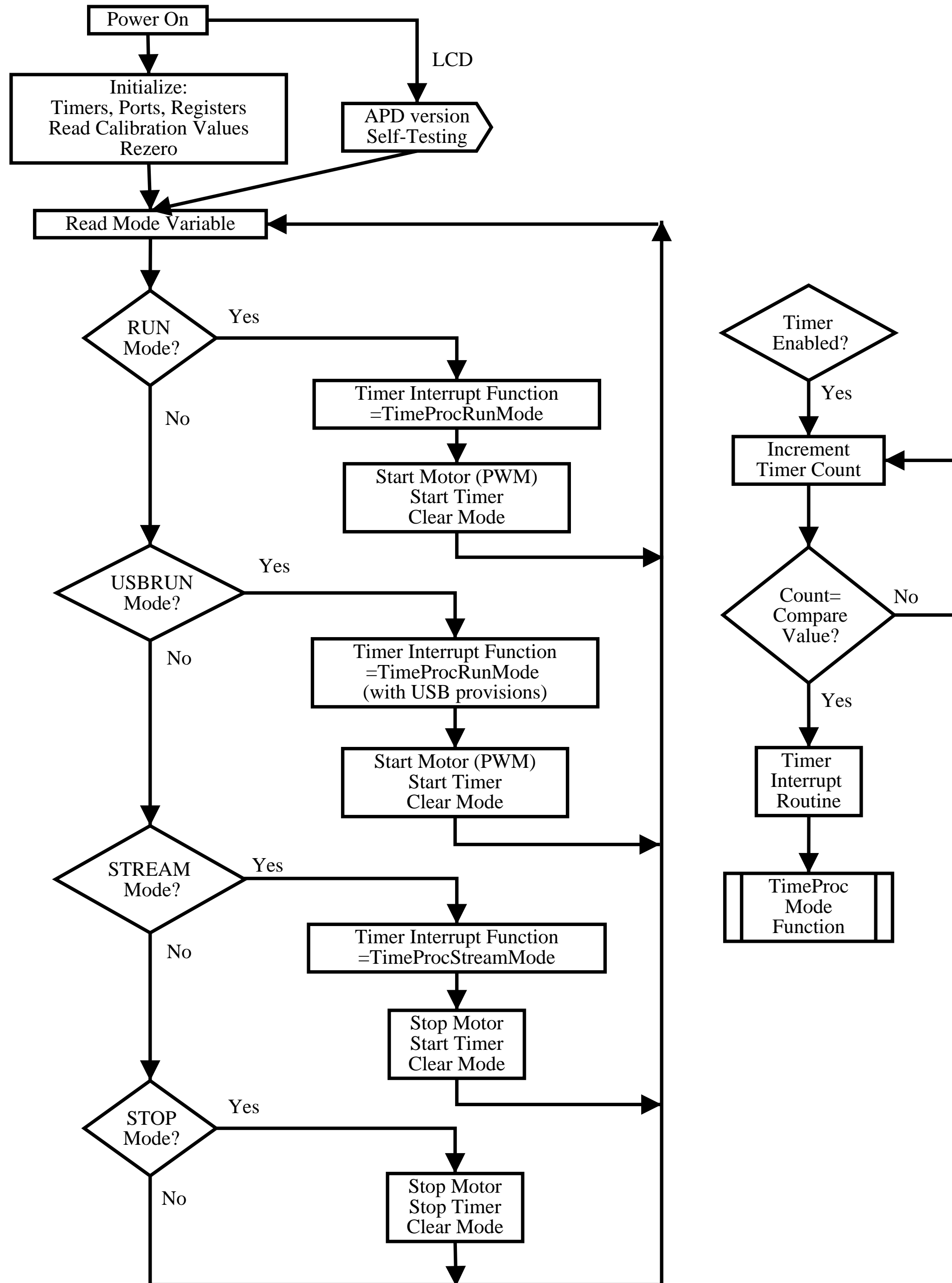


Figure 8. Diagram of the APD-SA DSP firmware polling loop that toggles among operating modes and enables timer interrupt routines to stream raw data or measure RR

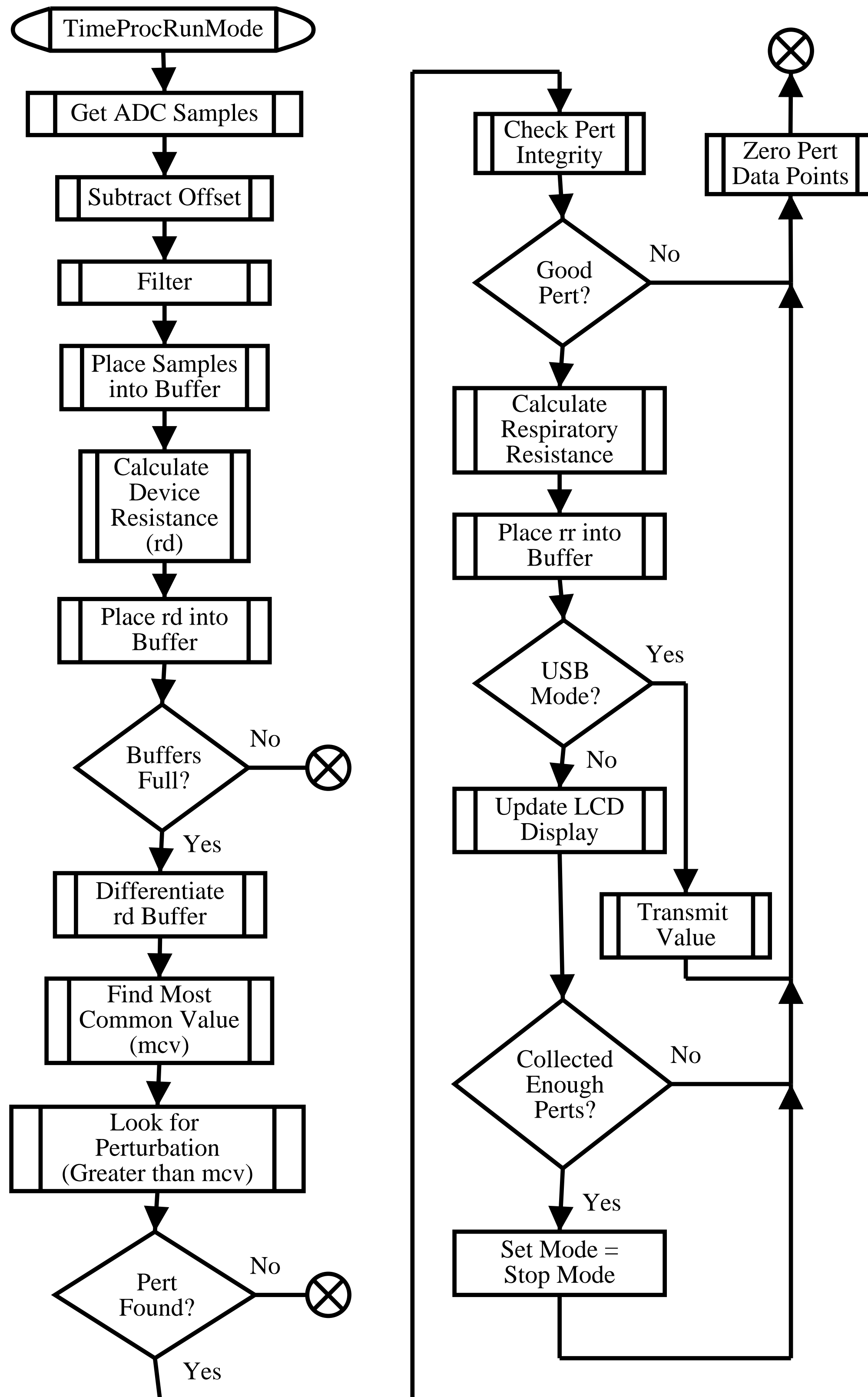


Figure 9. Diagram of the APD-SA TimeProcRunMode timer interrupt routine that triggers at the sample rate to measure pressure and flow to calculate RR

Software Description

APD200 functionality that supports the APD-SA can be considered to comprise two layers of software. The first layer provides a set of functions that mimics the APD100 capabilities, but was actually developed to work with the unique characteristics of APD-SA communication and data transfer. APD-SA characteristics that are unique with respect to the APD100 include data buffer size, numeric formats, the order of transmitted variables and the necessary command sequence to control the APD-SA over USB. The second layer resides beneath the first and translates the modified functions into USB-specific IO operations through the HID driver. The HID driver is shipped with the Windows operating system (OS) because the HID class of USB devices follows all or a subset of particular and consistent behaviors dictated by the USB HID specification. Operation as an HID required that the APD-SA function be constrained within the HID specification, but circumvented the complexities of developing a custom USB driver.

The APD-SA USB employs the EZUSB's “Renumeration” ability to present itself as an HID. Enumeration is the process in which the host PC reads the descriptor table of the USB device in order to properly identify the supporting drivers it requires. Figure 10 illustrates the overall process that occurs as the APD200 first communicates with APD-SA USB. After first power-up, the EZUSB loads its permanent read-only memory (ROM) code that contains its identification code as a Cypress EZUSB device. Upon first USB connection to the APD-SA, the host PC reads the identification code from the EZUSB chip and searches its registry to load the corresponding Cypress

driver. If the driver has not yet been installed onto the computer, Windows will go through its usual process of notifying the user of the newly discovered hardware. When the driver is loaded, it provides basic USB test functions through Cypress test software. The driver can also load under application control custom firmware into the EZUSB RAM. In APD200, when the option to connect via USB to the APD-SA is selected, APD200 searches the Windows system to determine if a Cypress EZUSB device has been loaded. If it sees the EZUSB, it will invoke commands to download to the EZUSB the custom developed APD-SA USB firmware. APD200 will then reset the EZUSB chip and the chip will upon reset identify itself as an HID because this identification is coded in its firmware. Windows will load the HID driver and APD200 will test for proper communication with the APD-SA USB.

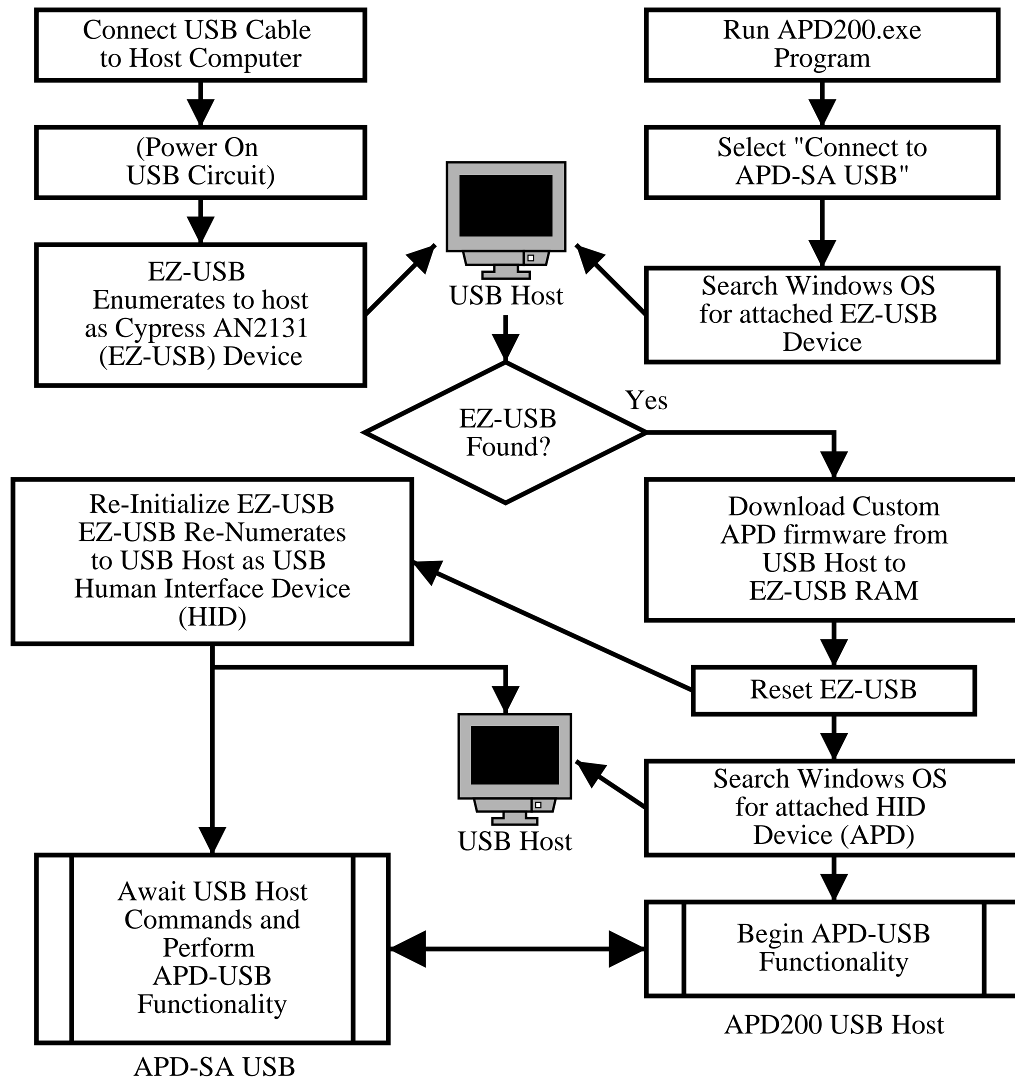


Figure 10. Diagram of the APD-SA implementation of the EZUSB "Renumeration" feature when APD200 invokes USB communication; USB cable connection (top left) and APD200 program execution (top right) are independent of each other and can occur in any order; APD200 software has provisions for both the presence and absence of the APD-SA USB connection

PROCEDURES

PC Hosted Prototype Development

The first step in APD-SA design was the evaluation of hardware and software options in a PC hosted prototype. APD code designed for a simple, fast-response APD-SA was written in MS Visual C++ using only C-compatible language. The PC executed the code in order to simulate an embedded processor. The PC communicated via parallel port with an external breadboard that contained the analog signal conversion circuitry, display circuitry and motor control circuitry. The breadboard was connected to the APD body to collect analog signals and control the motor. The code was optimized to minimize code size and maximize code speed. MS Visual C++ includes a time-profiling feature in which it monitors the time it takes for a section of code to execute. This feature was used to maximize code speed by comparing alternate methods of achieving the same result and selecting the fastest method. The prototype was used to estimate the processing and memory requirements of a stand-alone system. The design was also reviewed to determine the suitability of a fast-response device and the best option for a user interface. A detailed description of the PC hosted prototype can be found in Appendix I.

Integrated Hardware and Software Development

Following review of the first PC-based prototype, work continued with the development of the APD-SA on embedded processors. This work required EZUSB firmware, DSP firmware and APD200 software development on a PC and the integration of embedded processor development platforms with custom APD-SA hardware. There were many steps in this process.

The PC-hosted C/C++ prototype code was ported to the DSP56F803E. Code was written to initialize and control functional subunits on the DSP including PWM for motor control, RS232 communication, timer interrupts and external hard interrupts. The interrupt functions required proper interrupt vector table setup. DSP firmware required the adaptation of the C/C++ conventions to the Metrowerks IDE, altering some of the ways in which parameters were passed to the functions. The C code size had to be decreased to fit into the DSP FLASH and RAM. Variable types and buffer sizes were altered to a great extent in order reduce memory required. For example, pressure and flow data was stored in integer buffers. Debugging options were minimized. The alteration of variable types required the re-writing of some routines, as well as the modification of the application of calibration constants. Routines were written in assembly code to convert data types. A template memory map for the DSP56F803E was modified to provide space for calibration constants. Numerous details of the code adjustment can be reviewed in the code listed in Appendix D.

Once C code was ported to the DSP platform, the APD measurement hardware was integrated for testing with the DSP. ADC and motor control options were explored. ADC timing was adjusted. An RS232 PC terminal emulator (MS Hyperterm) was used to simulate the APD-SA display and monitor debug data. After several revisions of the hardware and DSP firmware, a final arrangement was selected. ADC data collection integrity was verified by logging data over RS232. The RR measurement routines were similarly validated.

After verifying the integrity of ADC and DSP performance, the EZUSB platform was integrated with the DSP and ADC. Digital logic to interface the EZUSB platform to the DSP and ADC was designed and tested. To more permanently join the platforms, corresponding connectors were built or obtained and modified.

EZUSB firmware, DSP firmware and APD200 USB software co-development ensued. The code listings contain comments that describe details of the EZUSB firmware and APD200 USB software development. Appendix E is USB firmware code and appendix F is APD200 code. Fundamental APD200 USB HID routines were modified and expanded for APD-SA-specific operations. Basic data streaming and variable exchange routines were developed on both the EZUSB and APD200 platforms simultaneously, followed by more complex RR measurement data transfer routines. DSP firmware was modified to respond appropriately to the possible timing and data transfer requirements of USB connection. Communication routines were developed to confirm command requests and pass data among DSP, EZUSB and PC. These routines required that a custom structure for APD-SA data streams be developed. EZUSB code was designed to keep intact this structure while managing the USB buffers. APD200

routines were designed to obtain these structures and extract variables and command codes from the streams. Details of command codes, buffers, data stream formats and data stream structures are described in the code listings. EZUSB debug information was passed to the host PC on a second RS232 channel, often connected to second PC. The EZUSB RS232 signals were first routed through an RS232 level-shifting chip, which was retained in the final APD-SA design to facilitate EZUSB debugging in future development.

Following ADC, logic, DSP, EZUSB and APD200 development, the LCD user interface for the APD-SA was integrated into the design. The DSP code was modified to include control routines for the LCD screen. The DSP code modifications included adjustments to the particular type of information that would be passed to the LCD screen. Appendix C contains tables listing the interconnections between hardware devices in the prototype. The tested prototype system including ADC, logic, DSP, EZUSB, LCD and APD200 was then reviewed for the suitability of its overall function.

Upon prototype design acceptance with some revision, a schematic layout was developed for a custom ADC, logic and power distribution PCB. Appendix B contains PCB layout images. The PCB positive image was printed on laser jet transparency film. A positive exposure kit was used to etch the layout onto copper clad board. Two copper clad boards were etched – one for the top and one for the bottom side. The boards were assembled and holes for components were drilled. IC and connector sockets, regulators and passive components were soldered. ICs were inserted into sockets. Performance of the PCB was tested. Some modification were performed by cutting traces and soldering jumper wires.

Once the custom PCB was determined to function properly, an enclosure to house all APD-SA electronics was developed. The enclosure was designed in CAD to be as small as possible while housing all electronics. Connector clearances and PCB dimensions were considered in the design. An orientation of PCBs was found that would minimize enclosure size, allow all inter-PCB cables to be connected and facilitate future debugging. Appendix A contains the CAD drawings for the enclosure. Upon assembly and testing, the APD-SA enclosure was ready for validation by calibration and human subject tests.

APD-SA Performance Validation

Calibration Performance

To validate basic data transfer and the accuracy of pressure and flow data in the APD-SA USB, calibration data for the APD-SA and APD100 were compared. For each device, three pressure and three flow calibrations were performed. Pressure calibration points were at zero and 10 cmH2O and flow calibration was performed for three Liters of air in five seconds for each flow direction, injection and withdrawal. The experimental calibration values were compared with theoretical calibration values derived from the specifications for the APD pressure sensors, PT, circuitry and ADC.

A statistical comparison of the calibration values was not performed, because the statistically different calibration constants between the devices does not imply different RR measurements – the purpose of calibration constants is to correct for signal transformation differences between devices. For example, if the two devices yielded statistically different calibration constants due to differences in circuitry, they could still yield accurate RR measurements because the calibration constants should compensate for the differences in circuitry. A qualitative analysis of calibration data was performed, however, to demonstrate the overall validity of APD-SA data acquisition.

Human Subject Testing

To examine the accuracy of the APD-SA, APD-SA and APD100 RR measurements were performed for human subjects. The format used here for APD measurement is identical to that used in prior and on-going APD measurements and thus falls under IRB approval 03-0324. Thirteen subjects older than 18 years of age were tested. For each subject, one APD100 measurement and one APD-SA measurement was made. The order of measurements was randomly determined and measurements were obtained within 15 minutes of each other. For the APD-SA measurement, the device was used in its USB mode to provide more detailed data. The APD body was placed on a ring stand and when an APD-SA measurement was required, the APD-SA enclosure was placed atop the APD body. Prior to each measurement, the APD was rezeroed, and prior to each APD-SA measurement, the device power was cycled, *i.e*. the power connection was removed and reattached.

Subjects were encouraged to find and maintain for both measurements, a comfortable sitting position. The height of the APD mouthpiece was adjusted to meet the subject's mouth once the subject was comfortably seated. The subject was instructed to hold the cheeks firmly during the measurement and breath normally through the device. Each subject used a disposable cardboard mouthpiece and re-usable, washable noseclip for the measurement set. For each measurement, at least 100 good perturbations were collected in each breathing direction. If during a measurement an error such as a USB miscommunication or perturbation wheel seizure occurred, the measurement was performed again after rectifying the problem. Average RR data for the APD-SA was compared with that for the APD100.

Further Investigation of RR Calculation Algorithms

During APD-SA firmware and hardware development, synthesized data sets had been used to test perturbation detection and RR calculations. These data sets had been constructed based on typical pressure and flow magnitudes obtained from previously collected APD data. However, a sampled data set had not been tested in both devices.

After human subject data collection and analysis, it was desired to briefly validate the APD-SA and APD100 RR calculation algorithms by using the same pressure and flow data set for each. Pressure and flow data during perturbations was collected on the APD-SA in JTAG debug mode so that execution could be paused and data buffers examined, as had been the process during early development stages. The calculated RR values for the APD-SA were recorded, and the pressure and flow data

points were transferred to a lookup table for the APD100. The APD100 software was then temporarily altered to read data from the lookup table to replace data collected from the DAQ. The APD100 RR calculations were then recorded. Algorithms were compared and discrepancies explored.

The PT heater was disconnected on the APD body for this research and not included as a design parameter for several reasons. The PT is heated to eliminate condensation and to better stabilize temperatures for volume correction. Proper inclusion of the PT heater would have required a good deal of additional design effort and may be better suited for future development. The PT heater requires nearly one watt of power depending on the desired temperature. This power requirement might be considered excessive for a device destined for hand held battery-operated use, though high-drain batteries are now available. The heater time constant is also a complication in implementing the heater because the hand held device is designed for quick start and measurement. In practice, PT operating temperatures are often achieved by allowing the PT to heat for a minimum of 10 to 15 minutes before measurement. Miller et al. (1997) provide a discussion of the impact of PT temperature on measurement accuracy, emphasizing its impact on volume calculation. Volume calculation was not studied here. For device validation, the heater was disconnected so that the measurement conditions would be identical for both the APD-SA and APD100.

RESULTS AND DISCUSSION

Design Results

The design process yielded a prototype APD-SA USB device capable of measuring RR as a stand-alone unit, and capable of providing more detailed RR data via USB connection to a host PC. Figure 11 shows the working APD-SA USB prototype with DC power connected. Figure 12 shows a subject using the device as a stand-alone unit to measure his RR. Figure 13 shows how the LCD screen on the APD-SA USB communicates device status and measurement progress to the user during measurement and upon USB connection to a host PC.

Figure 11. APD-SA USB ready to make an RR measurement

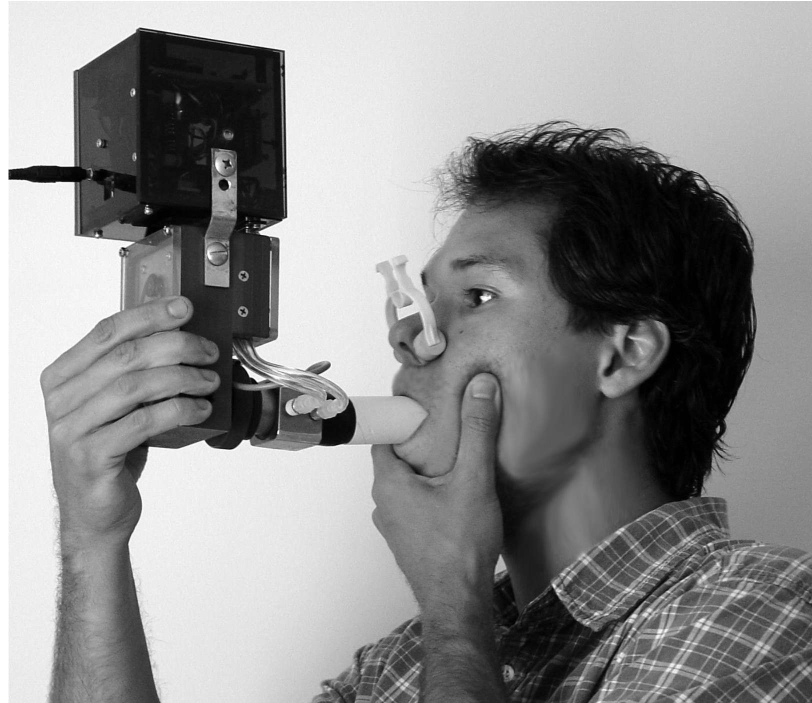


Figure 12. A subject using the APD-SA USB as a stand-alone device to measure his RR

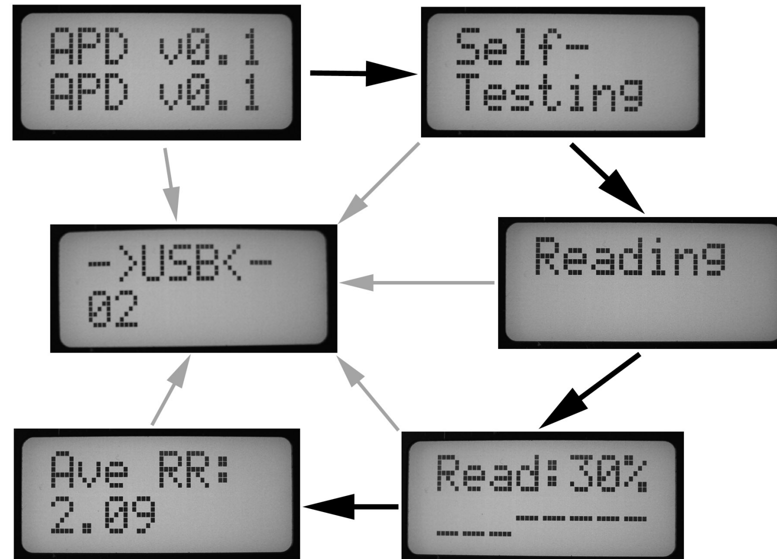


Figure 13. LCD display sequence on the APD-SA USB; clockwise from the top left: two start-up screens followed by screens that indicate to the user that the device is reading data; “Read:” followed by a percentage indicates to the user how much of the measurement has been completed, as a percentage of the total number of required good perturbations; the dashes at the bottom of the “Read:” screen move from left to right with every few good perturbations and the displayed percentage increases in 10% intervals; when a sufficient number of perturbations have been acquired, the screen displays the average RR; at any time during operation, a USB connection to the device will be indicated with the “USB” screen

APD200 software, a revision of APD100, expands upon the basic functions of APD100. Figures 14 through 17 present typical program windows in the host PC APD200 program when the software is used in conjunction with the APD-SA USB. The main APD panel screen (Figure 14) provides menus and data display during RR measurement. APD200 menu items include file functions such as saving data, USB connection functions (Figure 15), calibration functions (Figure 16), and data plotting functions (Figure 17).

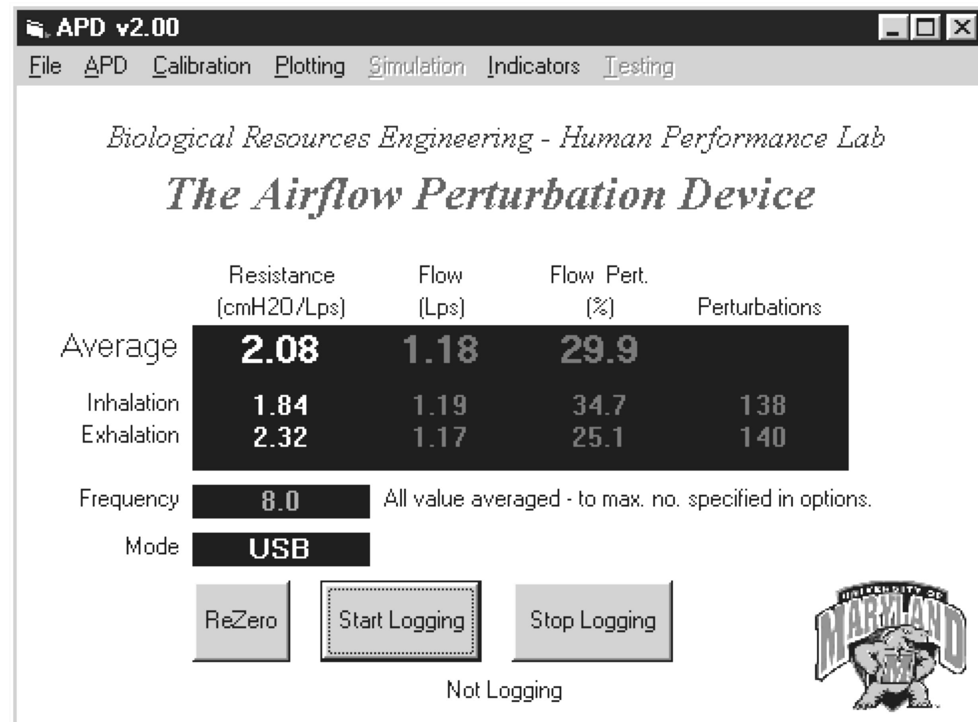


Figure 14. APD200 main panel window with menu functions and RR data display, shown here in operation with the APD-SA USB device

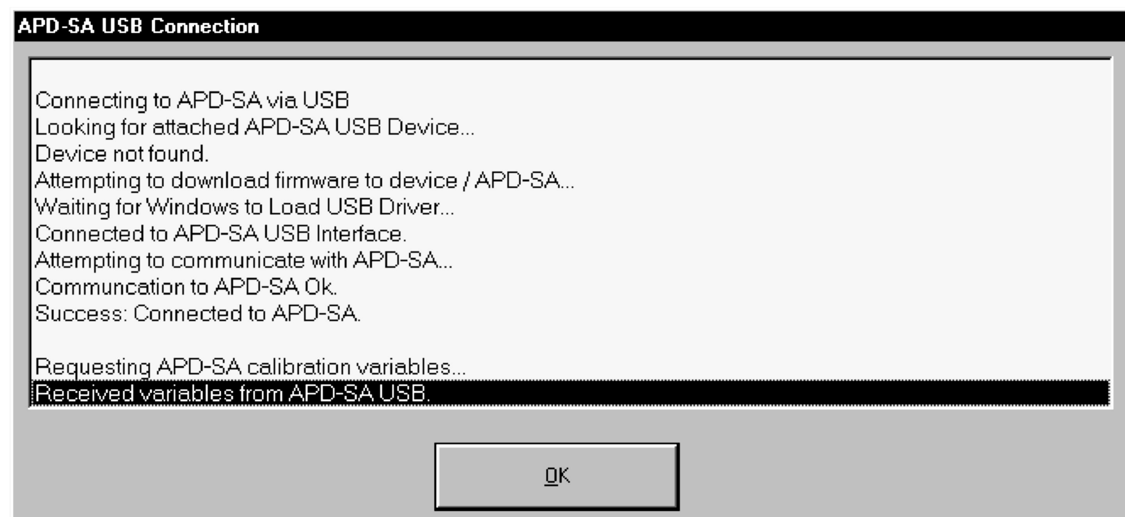


Figure 15. APD200 window that indicates to the user the status of the connection to the APD-SA USB when a USB connection to the device is first attempted

Figure 16. APD200 window that shows calibration constants, shown here in operation with the APD-SA USB; the window includes buttons to transfer calibration constants to and from the APD-SA USB

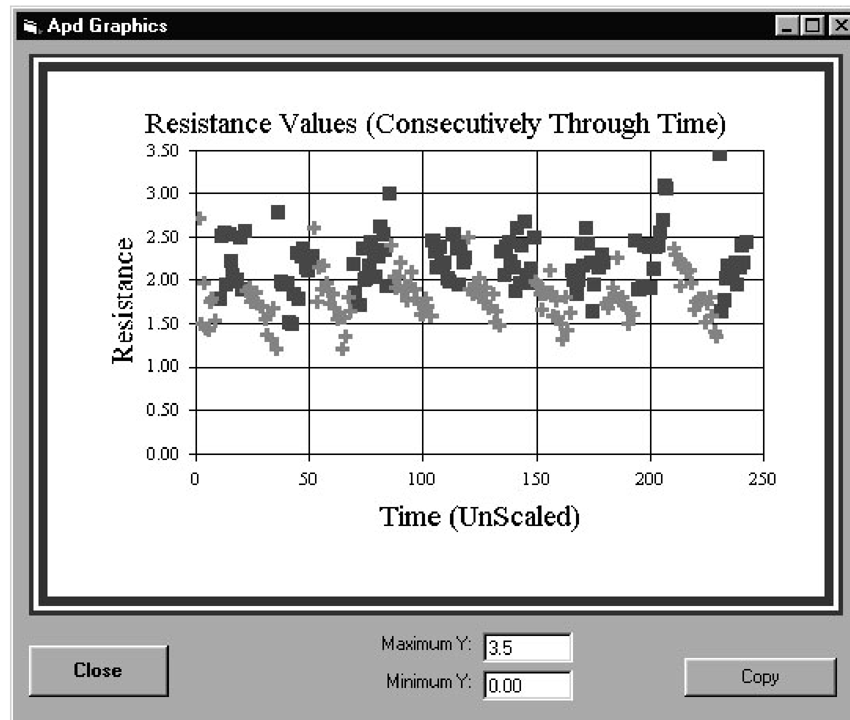


Figure 17. APD200 window that shows a plot of individual RR values over time measured by the APD-SA USB

Calibration Results

The calibration results demonstrate that the APD100 and APD-SA provide similar calibration values. The results demonstrate the integrity of the APD200 calibration procedure, the signal conversion and data processing in the APD-SA, and the APD-SA to APD200 USB communication. If any of these data processing and transmission steps were erroneous, then calibration values would greatly differ from the

expected values, and APD200 performance would differ greatly from APD100 performance.

Calibration span values are particularly important because the ratio of pressure to flow span is what ultimately converts unscaled data to scaled data and thus determines the RR reported by the device. Table 3 presents the measured flow and pressure spans for the APD100 and APD-SA as well as average, standard deviation and standard deviation as a percent of the average. Notably, all calibration span standard deviations are less than one percent of the mean value. The relative magnitude of flow and pressure standard deviations in the APD100 are similar to those for APD-SA.

Table 3. Span calibration results for APD100 and APD-SA; APD100 collects and scales voltage data while the APD-SA collects and scales ADC counts

APD100	Flow Spans		Pressure Span
	Inh (Lps/V)	Exh (Lps/V)	(cmH2O/V)
	-4.216	-4.392	-12.386
	-4.206	-4.379	-12.565
	-4.203	-4.340	-12.540
Average	-4.208	-4.370	-12.497
Std Dev	0.003	0.020	0.034
Rel Std Dev	0.07%	0.46%	-0.27%
APD-SA	Flow Span (Lps/count)		Pressure Span (cmH2O/count)
	-0.001312		-0.003843
	-0.001307		-0.003826
	-0.001307		-0.003846
Average	-0.001309		-0.003838
Std Dev	0.000001		0.000010
Rel Std Dev	0.08%		0.26%

To examine the similarity of calibration values between the two APD versions, it is useful to compare how each measured calibration value relates to its predicted value, and to examine the typical ratio between pressure and flow span. Table 4 presents the theoretical flow and pressure spans and a comparison between APD100 and APD-SA calibration span values. For the APD100, the theoretical span values were calculated based on manufacturer specifications for the No. 2 Fleisch pneumotachograph and the Honeywell Pressure Sensors. For the APD-SA, the theoretical span values were calculated based upon the same sensor specifications as well as the specifications for the ADS7825 16-bit ADC. Specifications for signal transduction elements are shown in Table 5. Typical theoretical calibration spans based upon sensor specifications are shown for APD100 and APD-SA in Eq. 1 and Eq. 2 respectively.

$$\begin{aligned} &Span_{Flow,\,APD100} = \\ &= \frac{3.253 - 0.346\,Lps}{1.0 - 0.1\,cmH2O}\, 2.54\,\frac{cm}{inch}\;\; \frac{1\,inH2O}{4.25 - 2.25\,VDC} \\ &= 3.230\,\frac{Lps}{cmH2O}\, 1.27\,\frac{cmH2O}{VDC} = 4.102\,\frac{Lps}{VDC} \end{aligned} \qquad \text{Eq. 1}$$

$$\begin{aligned} &Span_{Flow,\,APD-SA} = \\ &= 3.230\,\frac{Lps}{cmH2O}\, 2.54\,\frac{cm}{inch}\;\; \frac{1\,inH2O}{4.25 - 2.25\,VDC}\;\; \frac{20\,VDC}{2^{16} - 1} \\ &= 4.102\,\frac{Lps}{VDC}\;\; \frac{1\,VDC}{3276.8\,ADC\,count} = 0.001252\,\frac{Lps}{ADC\,count} \end{aligned} \qquad \text{Eq. 2}$$

Similar calculations yield theoretical calibration spans for each device as shown in Table 4.

Table 4. Data for a comparison of calibration results for APD100 and APD-SA, showing the measured and theoretical spans and percent difference between theoretical and measured values for each span; pressure to flow span ratio is the pressure span divided by flow span, the ratio that converts an unscaled respiratory resistance value to a scaled value

Theoretical Calibration Values		
APD100	Flow Span (Lps/V)	Pressure Span (cmH2O/V)
	-4.102	-12.700
APD-SA	Flow Span (Lps/count)	Pressure Span (cmH2P/count)
	-0.001252	-0.003876
Percent Difference Between Experimental and Theoretical Spans		
	Flow	Pressure
APD100	4.13% (avg inh exh)	-1.26%
APD-SA	4.39%	-0.77%
Ratio of Average Pressure Span to Average Flow Span		
APD100	2.936	
APD-SA	2.943	

Table 5. Specifications relevant to calibration span for sensing and data conversion components on the APD100 and APD-SA

APD Version	Part	Calibration Span-Related Specifications	
APD100 & APD-SA	No. 2 Fleisch PT	cmH2O	Lps
		0.1	0.346
		0.5	1.733
		1.0	3.253
APD100 & APD-SA	Flow Pressure Sensor Honeywell: DC001NDR5	±1 inH2O input across 0.25 to 4.25 VDC output	
APD100 & APD-SA	Mouth Pressure Sensor Honeywell: DC010NDR5	±10 inH2O input across 0.25 to 4.25 VDC output	
APD-SA	ADC: Burr-Brown ADS7825	±10 V input across 16 bits as twos-complement (0 to 10 VDC = 0x0000 to 0x7fff)	

Since both the APD100 and APD-SA incorporate the same PT and pressure sensors, it would be expected that excepting data processing errors, calibration and measurement should yield similar results. The APD-SA does however include unity-gain buffering circuitry to drive the ADC inputs, and thus may introduce some undesired signal transformation prior to the conversion. In the APD-SA PCB layout, the buffer amplifier supply voltage is 5 VDC, thus a non-linear or even saturated response may characterize the collected data for large sensor output values near 3.8 volts (see the specifications for the buffer amplifier). The APD is typically calibrated at pressures and flows that would cause the sensors to produce voltages less than this saturation voltage. The 3.8 VDC saturation corresponds to about 6 Lps flow or 18 cmH2O pressure. Typical values during measurements have been less than 2 Lps in the highest flow subjects and less than 3 cmH2O pressure during perturbations. Pressure calibration is typically conducted at 10 cmH2O and 3 Liters volume injected over 5 seconds. Nonetheless, it is possible, though not likely, that large perturbations causing large pressures and to a lesser extent, high flows, might induce large sensor output voltages that would enter the troublesome buffer amplifier input voltage range. Thus APD calibrators should avoid fast volume injections, and RR measurements should be made at proper perturbation magnitudes.

Subject Testing Results

Subject Summary Data

RR data was collected for 13 subjects. Subjects were between the ages of 18 and 60 years of age. Six subjects were male and seven were female. Subjects ranged in height from 1.57 to 1.88 meters and in mass from 55 to 105 kg. Of the subjects, two were smokers, one heavy, and two had mild asthma that was inactive at the time of measurement. Two measurements had to be repeated due to perturbation wheel seizure, one of which also had a USB communication failure. In general, subjects did not report any large discomfort during measurement. Two subjects suggested creating an arm that would hold the APD over the edge of the table for easier posture during measurement . One subject reported that holding the cheeks caused her to feel as if her breathing were restrained because her arms were positioned against her chest.

Appendix G contains the detailed APD data and the random numbers used to determine which device was used first. RR values ranged from near 2.0 cmH2O/Lps to near 4.5 cmH2O/Lps. The largest value was obtained for a 45 year old subject who had been smoking since he was 18.

Preliminary measurements had been made allowing subjects to hold the APD in one hand while holding the cheeks with the other. However, some subjects, primarily those with smaller builds, rested their arms upon their chest and were reportedly more

muscularly tense while holding the larger APD-SA. Thus, the APD body was placed on a stand for all measurements.

APD-SA and APD100 Average RR Comparison

Figure 18 shows the relationship between average RR values from the APD-SA and APD100, one datum per subject. The data show that the APD-SA and APD100 return similar RR values. A statistically significant correlation of 0.98 with $p<0.0000001$ was found for the relationship between APD-SA and APD100 RR values. The linear regression slope was found to be 1.13.

To examine the practical meaning of the regression slope, it is useful to take an example. The regression returns an APD-SA value of 3.1 cmH2O/Lps for an APD100 RR value of 3.0 cmH2O/Lps. This difference is near one standard deviation of consecutive measurements with the APD100 (Lausted and Johnson, 1999). Thus the APD-SA returns values that are usable, but on average larger than APD100 values in a meaningful way, near one standard deviation for values around 3 cmH2O/Lps.

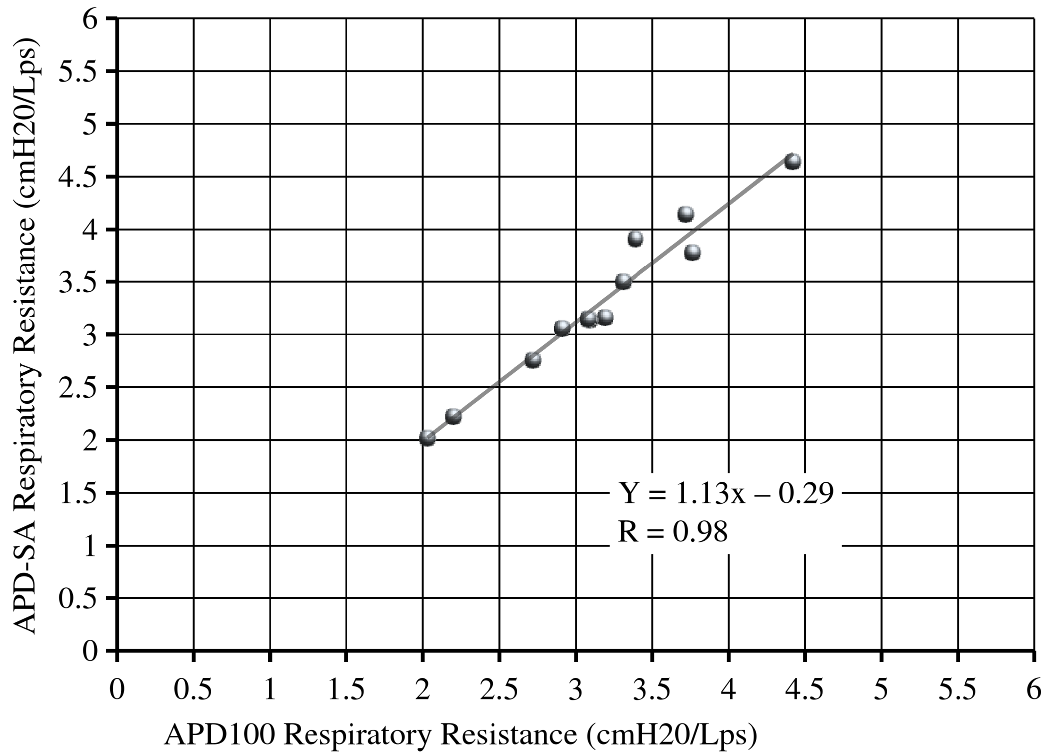


Figure 18. APD-SA average RR plotted against APD100 average RR for 13 subjects

Since both APD-SA and APD100 use the same PT and pressure sensors, differences between the two devices might occur in the amplification and data conversion stages for processing the data. There may be non-linearities in the APD-SA buffer amplifier response or conversion circuitry. Calibration is designed to obtain an average response for the sensors and circuitry as a whole, thus compensating for any differences between devices in data conversion. It is possible that in a non-linear data

collection response, calibration occurred at a large pressure value, but smaller pressure values obtained during measurement lie along the portion of the amplifier response that deviates most from the calibration's linear slope. Nonetheless, for there to be a

difference between devices in RR value, the ratio between pressure and flow would have to be different from one device to the next. To achieve this, one channel of data would have to deviate from a linear response in a convex manner while the other channel deviated in a concave manner. An examination of the effect of calibration values on the returned RR values might demonstrate whether typical calibration values alter RR enough such that non-linearities would need to be considered.

Investigation of Measurement Differences

The APD-SA's tendency to return larger RR values than the APD100 was investigated by examining the effect of calibration values. Since the application of calibration constants to data essentially scales the data, data from both devices was rescaled to reflect the effect of a calibration that would have caused the regression slope to become less. RR data from each device was divided by the ratio of pressure to flow calibration constants used for a measurement. Data points for the APD-SA were then multiplied by the smallest ratio of calibration constants that was obtained for either device, and the data points for the APD100 were multiplied by the largest ratio of calibration constants. A regression was then performed again, yielding a slope of 1.10. Thus even for calibration constants that would tend to sway the regression slope toward one, the APD-SA still exceeded APD100 RR measurements.

A second difference between the two devices' measurements may have occurred in the algorithms that calculated RR. The APD-SA perturbation detection algorithm was changed several times. Since the APD-SA uses a much smaller data buffer than the

APD100, it was possible to make the device much more sensitive to perturbations because perturbation detection parameters could be set for each perturbation rather than an entire data buffer. However, when preliminary data was collected for several subjects, the APD-SA returned larger RR values to a greater extent than data presented here. To make the measurements compatible with the large amount of data previously collected by the APD100 in laboratory studies, this sensitivity was decreased. After a second preliminary test of subjects, the detection algorithm was again altered to match the APD100 perturbation detection parameter method more closely. Though suggesting an over prediction of RR by the APD-SA, the results presented here represent the third revision of the APD-SA perturbation detection parameters. The preliminary tests, though not rigorously statistically tested, appear to show that RR calculation is sensitive to the perturbation detection algorithm, especially the manner in which the edges of the perturbation are selected. This is discussed further in the section: Suggestions for Further Research.

After data analysis, the APD100 and APD-SA device algorithms were again compared. The first major remaining difference in perturbation detection was the fact that the APD-SA sets detection parameters for each individual perturbation as it slides through its data buffer while the APD100 examines a five second data buffer, setting parameters based on the whole buffer. If device resistance (RD) changes with flow rate, which qualitative observation of the flow and pressure waveforms during measurement suggests, the perturbation detection parameters obtained from a large data set might tend to truncate perturbations that occur nearer the edges of breath. As evidence of this truncation, Yeh et al. (1982) present for a Fleisch No. 3 PT the non-constant

conductance increasing from 4.7 to 5.1 Lpm per cmH2O from near zero to 0.5 cmH2O pressure differential. This would only have impact on the RR value if pressure and flow do not have the same curvature during a perturbation. Again, qualitative observation of data indicates that inertia may play a role in the flow waveform.

A second major remaining difference in the perturbation detection methods is that the APD-SA detects perturbation edges slightly differently. Both devices use an RD threshold and derivative of device resistance (DRD) to select a perturbation edge. However, the APD-SA finds the beginning of the perturbation at the point where the RD is below the threshold and the DRD is positive, that is RD is increasing. The APD100 selects the beginning of a perturbation by using the same criterion that RD is less than the threshold, but looks for a point where the DRD is negative, that is where RD is decreasing. This may have a tendency in the APD100 to extend the assumed beginning of the perturbation edge beyond where the APD-SA algorithm defines it. Both devices define the end of the perturbation as the point where RD is below the threshold and DRD is negative (see Figure 19). The effect of this difference is not clear, and would require the examination in detail of the relationship between pressure and flow at the edges of a perturbation. For example, when moving along a perturbation from its center to its start backwards in time, if the point at which the RD rises occurs primarily from pressure as a result of flow phase lag, then the magnitude of the pressure perturbation might be underestimated.

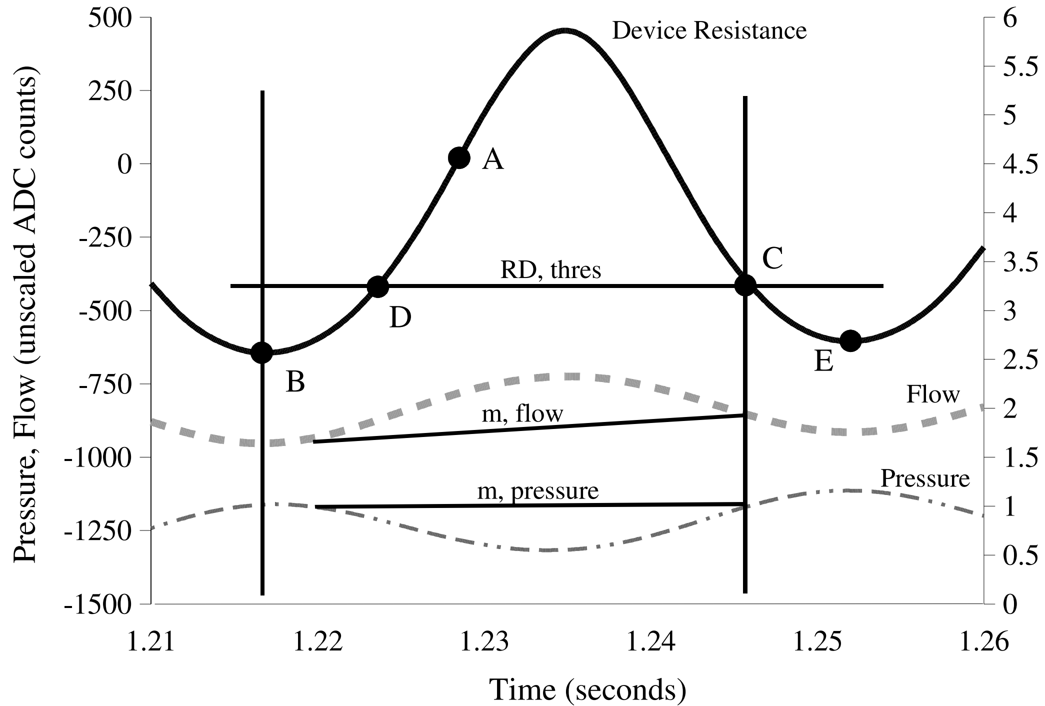


Figure 19. Device resistance, pressure, and flow during an inhalation (negative flow and pressure) APD perturbation; APD100 samples every several data points until one is found above an RD threshold (RD, thres); when the first point is found (A), APD100 then scans backwards for the first point at which RD is below the threshold and the slope of RD is negative (B); APD100 then scans forward for the first point having the same value and slope characteristics as (B) which is (C); RR is then calculated based on flow slope (m, flow) and pressure slope (m, pressure) between points (B) and (C); true RR would probably best be calculated between (C) and (D) or (B) and (E)

In the test of an identical data set, differences between the APD-SA and APD100 measurement were found. APD-SA was found to give a larger RR value than the APD100. When the APD100 perturbation edge detection was changed such that it looked for increasing RD at the start of a perturbation, the RR value returned was identical to that found in the APD-SA. A more complete analysis along these lines would be required to draw substantial conclusions, but it appears that the different edge detection methods may influence calculated RR magnitude.

CONCLUSIONS

1. The APD can be packaged into a stand-alone device that collects and analyzes data, displays RR to a user, and offers additional functionality via USB connection to a host PC.
2. It is possible to configure the APD-SA USB for fast-response such that the LCD shows a weighted average of RR values that is updated with each new usable perturbation that is collected; however, the fast-response option is not appropriate for a basic RR measuring device.
3. The APD200 program successfully incorporates APD-SA USB calibration protocols and expands data collection to allow the storage of pressure, flow, and RR throughout a measurement.
4. The APD-SA USB provides usable RR measurement values that correlate with previous RR measurements; however, an uncertain parameter in the APD100 precludes direct comparison between the two.
5. Algorithm parameters used to detect perturbations may effect the calculated value of RR.
6. Anomalies, quirks, and deviation from documented behavior greatly complicate the implementation of semiconductor devices and semiconductor prototype PCBs necessitating familiarity with each family of semiconductor devices prior to implementation.

SUGGESTIONS FOR FURTHER STUDY

Explore APD Algorithms

The expansion of powerful MCUs in small packages facilitates the development of newer and more robust APD algorithms that are improved in the areas of perturbation detection and perturbation integrity checking. Throughout APD-SA development and testing possibilities arose to implement adjustments in the perturbation detection and respiratory resistance calculation algorithms. For example, it became clear that the APD could be made more sensitive by decreasing the threshold at which a perturbation was detected. However, preliminary work with different thresholds, both lower and higher, appeared to show slight differences in calculated RR values. An examination of the pressure and flow waveforms showed that phase delay and threshold might interact to produce these calculated RR differences. Since the RR calculation is based upon a ratio, it might be assumed that a larger detection threshold would have little effect on the RR calculation. A preliminary look at the pressure and flow waveforms again showed that they differ enough in their curvature to change the RR calculation. For example, flow might be analogous to a $\sin^2(t)$ while pressure might be analogous to sin (t) function during a perturbation. Further, flow waveforms appeared to differ from pressure waveforms at the edges of the perturbation. In general it appeared as though the flow waveform did appear to demonstrate inertial effects as might be expected.

Perturbations might be more carefully checked for integrity by examining their shape or proportions. For example, the point at which the virtual pressure and flow is calculated might be checked for its proximity to the perturbation edges. The variance of perturbations might be continually monitored to filter anomalous RR values. Perturbations of excessive length or flow reduction could be rejected and used as an indicator of potential malfunction to the user because they probably occur due to a slowing of the perturbation wheel when it binds to the APD body or becomes gummed with deposits. With the available PWM motor control units on the DSP, automatic perturbation magnitude control could now be implemented by providing feedback to a motor that controls the variable resistance collar on the APD body.

Improve APD Hardware and Software

APD System Expansion

The APD system could be expanded to include features that facilitate the management of many APDs. A standard format for summary resistance data and calibration files might be devised and used to create a database. A database could then automatically track calibration data for each device and the RR measurement data could be automatically generated, in addition to the hard copy paper backups. Such an arrangement would eliminate data entry steps and facilitate queries for study of the data.

Further, medical clinics have been moving in the direction of linking home medical devices over a network to databases for remote monitoring and data collection.

The APD system might be expanded to include abilities for gathering more types of information, and better using the information it does collect. Since motor speed is controlled by the MCU, a range of frequencies could be swept to collect RR data as a function of frequency or estimate other mechanical properties of the lungs such as compliance. The APD already collects flow data and estimates lung volume. This data could be used more extensively to generate as a standard part of the displayed summary data numerical values that might characterize each subject's RR at various lung volumes and flow rates, in addition to the plots it already provides. In a standard and automated output file for each measurement, this type of data could be automatically logged without display if required.

APD Software

As with any software and hardware development projects there are bugs both known and unknown in both APD100 and APD200 portions of the APD host PC software. As the project becomes larger, no doubt additional effort will be required for debugging over the long term. In general, it would probably be useful to overhaul the software to make sure it provides the functions that are most needed, and those that would be most helpful. Further, its response to errors could be improved to provide more user feedback and more automated error prevention. Eventually, the software will need to be ported to other operating systems which may require adjustments in the

drivers used by the software. A more specific list of issues for correction and expansion in each area of the APD software follows.

APD100 Software

In the FindPerturbations routine, the beginning of a perturbation is detected by moving from the middle of a peak in device resistance to a point earlier in time when device resistance falls below a threshold and the derivative of device resistance is less than zero. This is probably unintended, as this condition would make sense at the end of a perturbation but not at the beginning. In the beginning of a perturbation it would be expected that the derivative of device resistance is positive. The code was written in this manner in very early APD code as well in APDKM7.XLS, an MS Excel-based macro that performed the calculations. This should probably be changed, and its potential impact on measurements examined.

In the PertDetectParams routine, the resistance threshold, resThr, above which the software begins to look for a perturbation may exclude good data. The value of resThr is the sum of inhalation and exhalation thresholds. First, in the case when one breathing direction is not yielding usable perturbations, the threshold is set very high, and no perturbations are considered, even if they are valid in the other direction. Secondly, when this summation of thresholds is used it makes the device overall less sensitive to smaller magnitude perturbations even if they are valid. If one airflow direction through the device incurs higher resistance, then the measurement effectively becomes less sensitive to perturbations in the breathing direction with lower device

resistance. The is especially relevant for the case in which the APD perturbation wheel flaps back and forth with each breath, creating differing degrees of perturbation in each direction. In either case, the issue might be worth addressing.

APD200 Software

There are several areas for debugging and improvement in the APD200 software specific to APD-SA functioning and general APD PC host software improvement:

- Work through any bugs in window appearances, calibration file loading or expected performance.
- Perhaps automate the display and calibration file loading depending on the selected device. This might require expansion of the APD settings file to include names for each calibration file used in a session, not just the last one selected prior to shutdown.
- Fortify the USB functions. The USB functions interact extensively with the operating system and related application programming interfaces (APIs), and complications that hang the application or PC should be examined. Issues include USB bandwidth saturation, OS hang on USB disconnect and other issues that as of now require reboot to reinitialize the USB subsystem.
- Check USB functionality when a user already has an HID device attached to the system.
- Port USB function to other operating systems.

- Consider custom USB driver development. Custom USB driver development requires extensive testing with the operating system. Software to generate a USB driver template is available at a fairly substantial price.
- Develop a help menu that includes troubleshooting and configuration tips.
- Include messages about device operation to the user directly on the APD panel while the program is running.
  - Why perturbations are not being detected (low flow, insufficient magnitude).
  - Performance warnings such as unreasonable calibration values, ADC or DAQ saturation, likely perturbation wheel stall or binding.

APD-SA Performance in Firmware

- USB-related improvements:
  - Purchase a full EZUSB development platform to improve USB reliability. Since a full EZUSB development platform was not purchased, the debug versions of firmware had to fit into the EZUSB internal RAM, and debugging could only be accomplished through RS232 output. When code for all routines was written, there was little to no memory left for debug code and thus debugging stopped when the needed functionality was achieved, though infrequent miscommunication still occurred – usually after several RR measurement collection commands. It is advisable to reset the device after each measurement by unplugging both the USB cable and power.

- In the case of a USB miscommunication, implement a synchronization data frame to re-send data and indicate the point at which data is being re-sent. At present, a miscommunication will corrupt any future data transmission until device reset.
- Implement more efficient USB data transfer to allow higher usable data rates for the same bit rates. This might require, or at least would benefit from, the development of an external (.dll) file developed in C or C++ to handle bitwise operations and data manipulation. This type of data manipulation is possible in Visual Basic (VB), but initial tests found it unreliable and limited in its applicability.

- FLASH memory download and uploads:
  - Consider downloading from the APD-SA the device serial number and firmware revision when needed after future development.
  - Implement bi-directional flow calibration values. At present, an average calibration flow constant is used for both inhalation and exhalation.
  - Implement a bitwise mode constant that can be uploaded to select fast or average RR display modes, spurious data point rejection and the maximum number of collected perturbations. This would require the corresponding control panel in the APD200 software.
- Implement feedback control of perturbation rate. Initial code provisions have been made for this but require refinement.
- Implement the fast-response option. The fast-response option can update the APD-SA display with an RR value at each time a good perturbation is measured. Initial code was developed and tested to provide this functionality but has been commented

out and was not included in the final design. The initial fast-response code included provisions for average and weighted average filters in order to smooth the displayed RR value.

APD-SA Hardware Improvements

- Redesign a circuit and layout to incorporate all components into a minimum number of circuit boards designed specifically for the APD-SA. This would dramatically reduce the package required for the APD and decrease eventual manufacturing cost by eliminating unused components.
- Use custom LCD screen by writing firmware for an inexpensive MCU to control the LCD, or similar. Potentially buffer LCD-destined DSP output and have the LCD-devoted MCU read the buffer to reduce RS232 time delays.
- Use a larger supply voltage for the ADC buffer amplifier to allow the use of the full range of the pressure sensors. The supply voltage presently limits the range over which the buffer tracks sensor output. Further, since the full range of the pressure sensors is at present rarely used, more sensitive pressure sensors could be incorporated to provide a larger signal-to-noise ratio.
- Bring the LEDs on the USB and DSP PCBs to a place in which they are visible to the user. They may be very helpful in indicating USB connection and perturbation detection. At this point, the yellow-green LED on the DSP PCB toggles each time a good perturbation is detected, indicating proper performance to the user that patiently awaits measurement.

- Include the PT heater in the APD-SA design or use a different flow transducer. A tap from the DC power supply to the APD-SA could be used with a power resistor to provide DC current to the PT heater, but the operating current will be near the maximum for the power supply used in development here.

APD Body Mechanical Improvements

- Improve the perturbation wheel mechanism. It presently moves toward the body on inhalation and away on exhalation, creating a different degree of perturbation in each direction. The wheel also tends to bind to the APD body after many uses.
- Tighten belt and reduce drive-shaft pulley diameter to improve perturbation wheel control. Or continue seeking alternate closure methods to work around the higher torque at low speeds required for the perturbation control.

APPENDICES

LIST OF APPENDICES

APPENDIX A: APD-SA ENCLOSURE CAD DRAWINGS..92

APPENDIX B: APD-SA PCB LAYOUT IMAGES..98

APPENDIX C: APD-SA PIN CONNECTIONS..101

APPENDIX D: DSP FIRMWARE CODE...103

APPENDIX E: USB FIRMWARE CODE...200

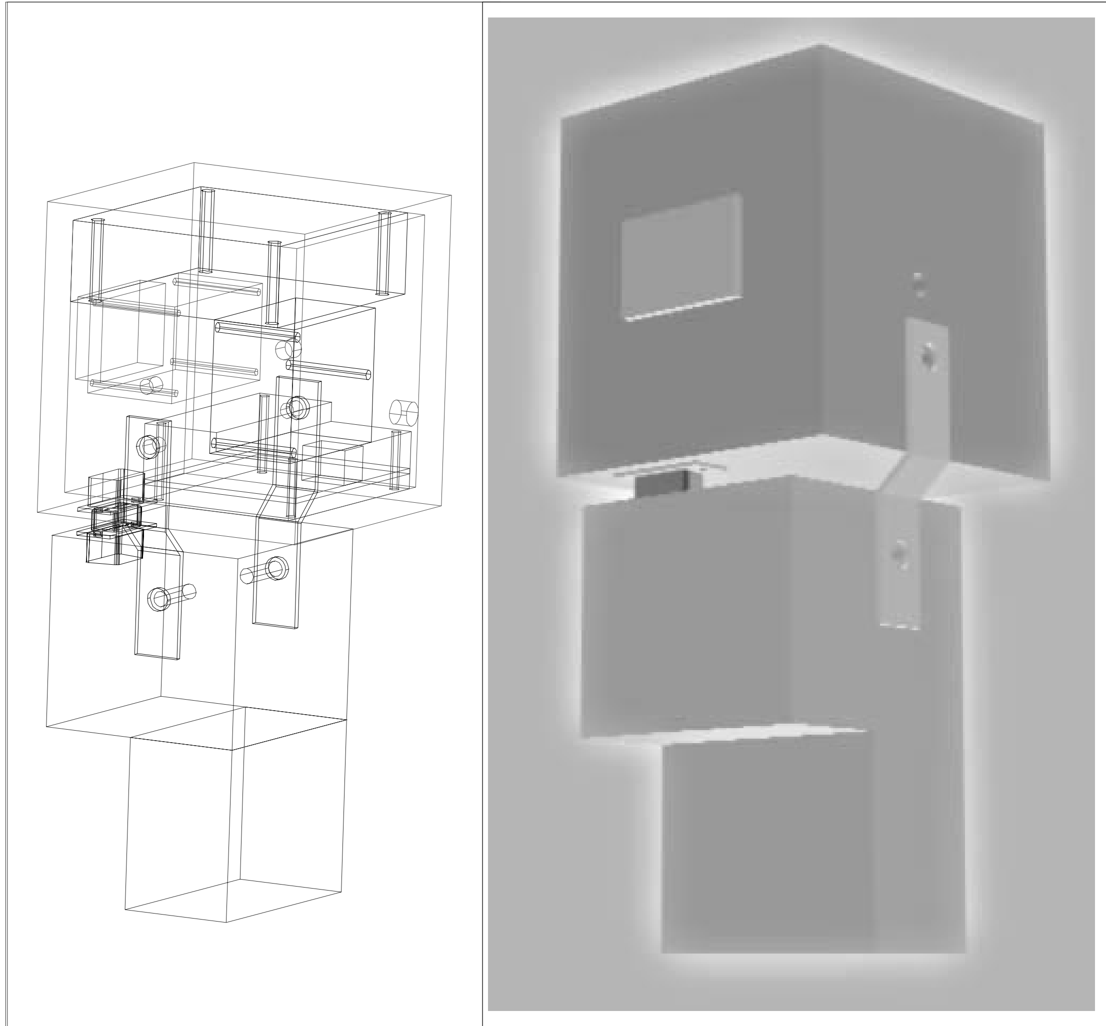
APPENDIX F: APD200 SOFTWARE – REVISED PORTIONS OF APD100...........233

APPENDIX G: SUBJECT DATA..303

APPENDIX H: EQUIPMENT AVAILABLE FOR APD-SA DESIGN......................305

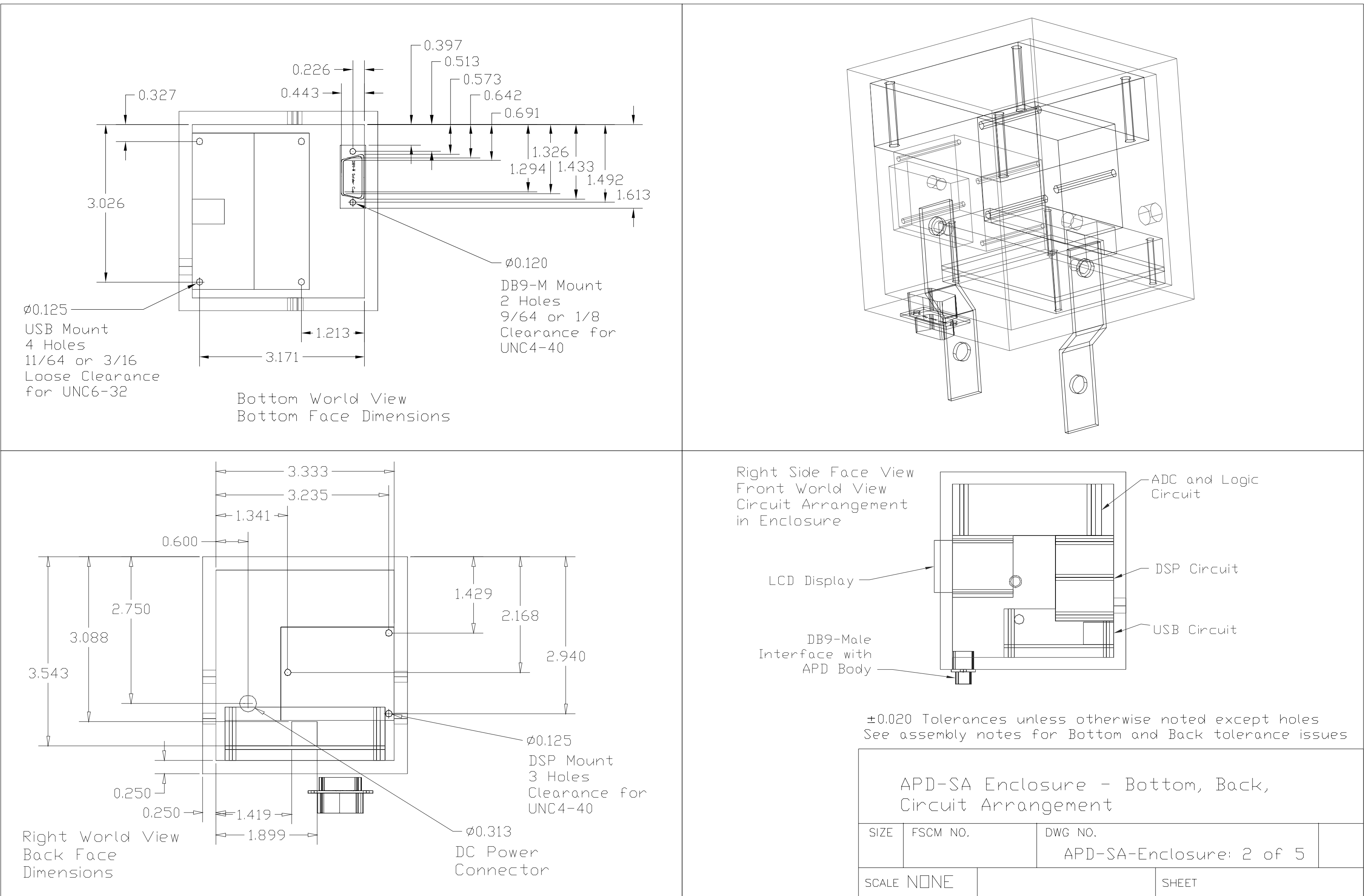
APPENDIX I: PC HOSTED PROTOTYPE...326

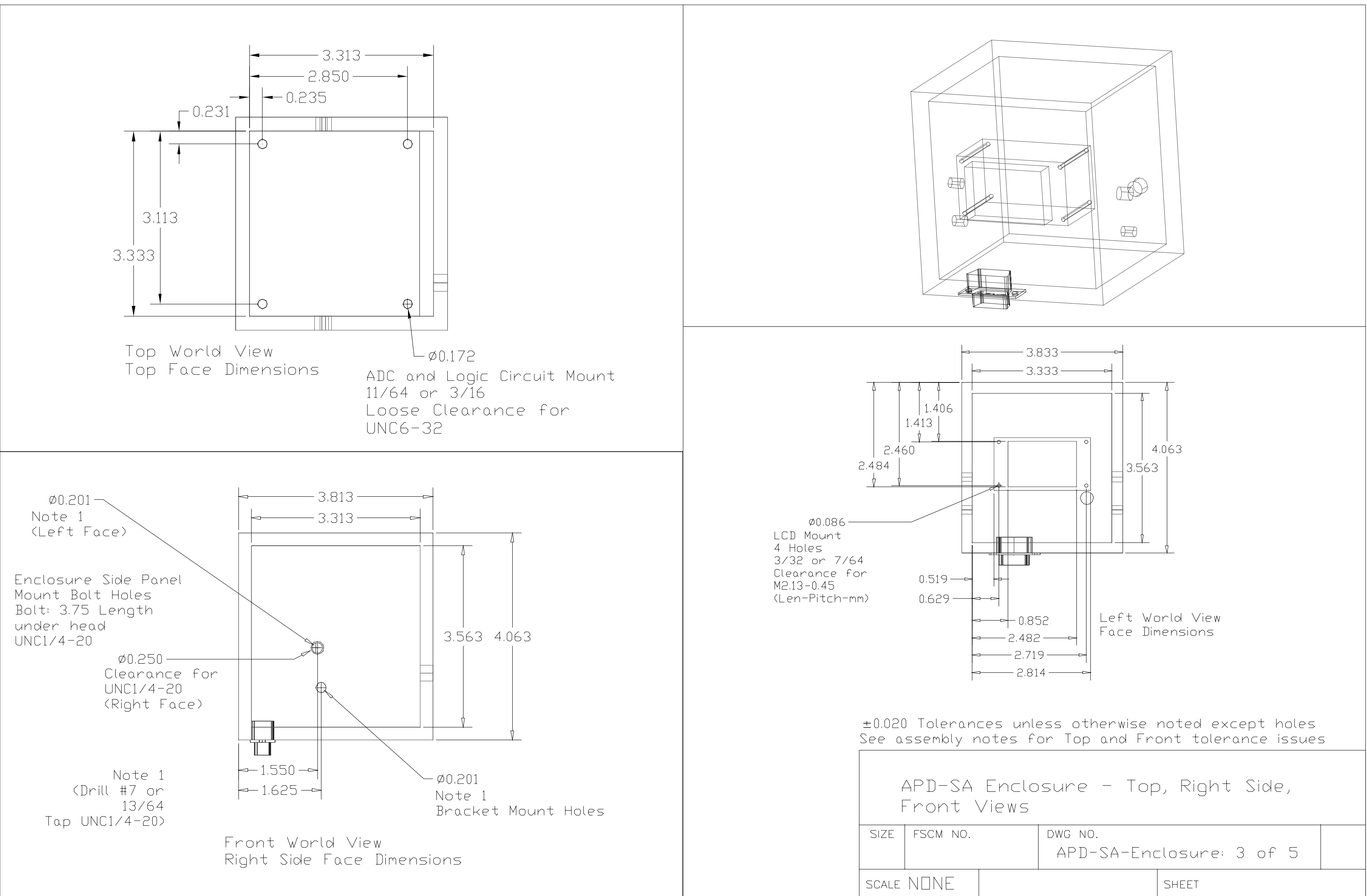
APPENDIX A: APD-SA ENCLOSURE CAD DRAWINGS

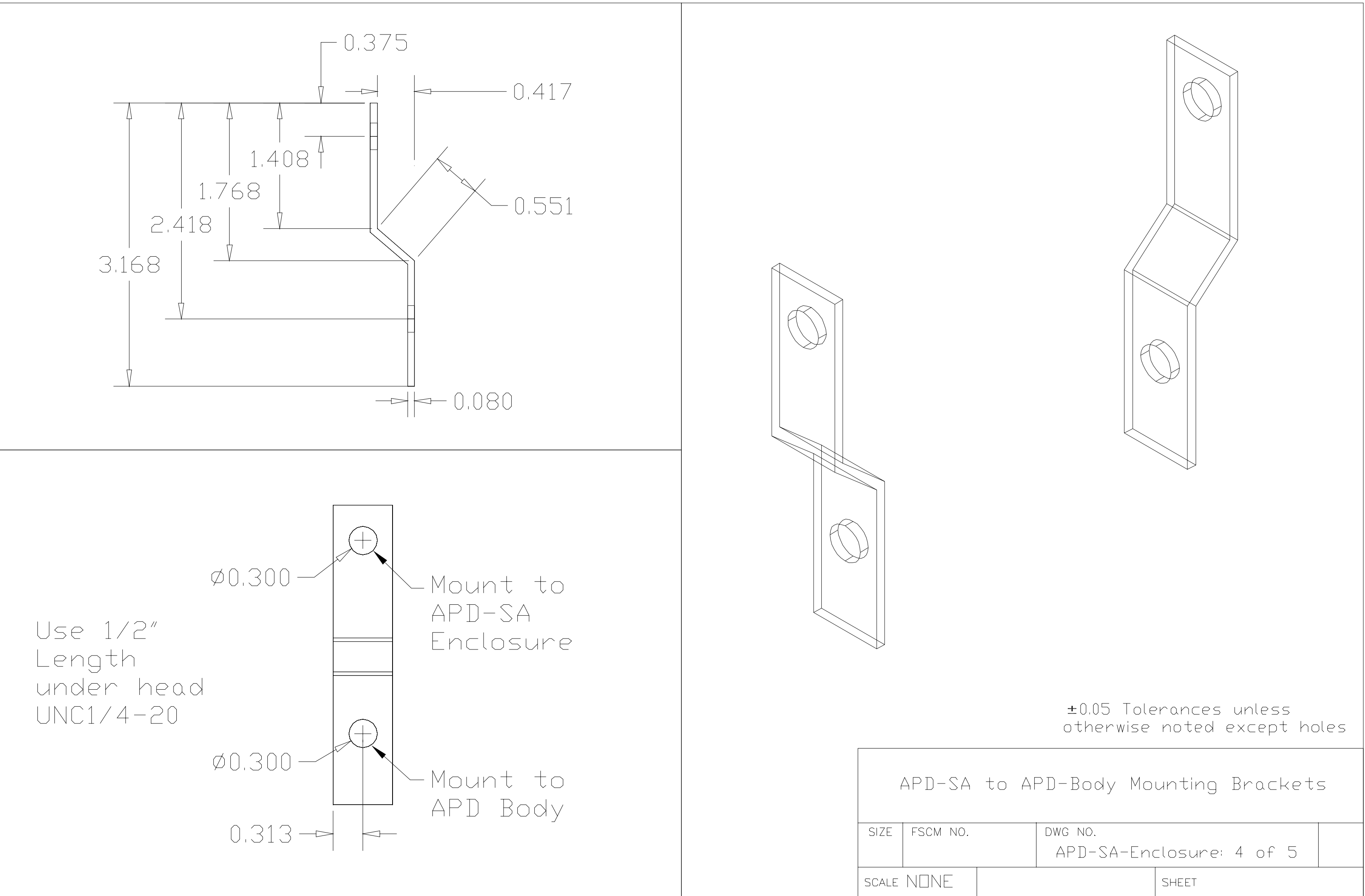


General Assembly and Prototype Notes:
* APD-SA Enclosure designed to mount onto existing design for APD Body previously used with computer-based DAQ
* Prototype enclosure assembled from 1/4-inch thickness cell cast acrylic plastic sheet
* Enclosure faces:
  a. Top, Bottom, Front and Back faces all cut from single width of plastic to ensure identical widths (cut with router or milling machine for fine edge finish).
  b. Top and Bottom faces are shortest and sit within the front and back faces
  c. Top, Bottom, Front and Back faces glued together with Weld-On #3 Acrylic glue (Methyl Chloride and other VOCs) - used eye dropper to saturate edge joints and left compressed for a day.
  d. Sides were cut to fit outside of the Top, Bottom, Front and Back faces and were fastened to the enclosure by inserting the UNC1/4-20 bolt through the enclosure between side faces.
* Circuit mounting: circuits were mounted inside of the enclosure once the Top, Bottom, Front and Back faces were glued. Except for the LCD circuits, all were interconnected with connectors, ribbon cables etc. and then slid into the enclosure and fastened with screws. LCD was mounted first and then electrically connected once other circuits were mounted.
* DB9-Male connection: all wires from circuits were drawn out through the DB9 cutout and soldered to the DB9. Then, DB9 was screwed into place.

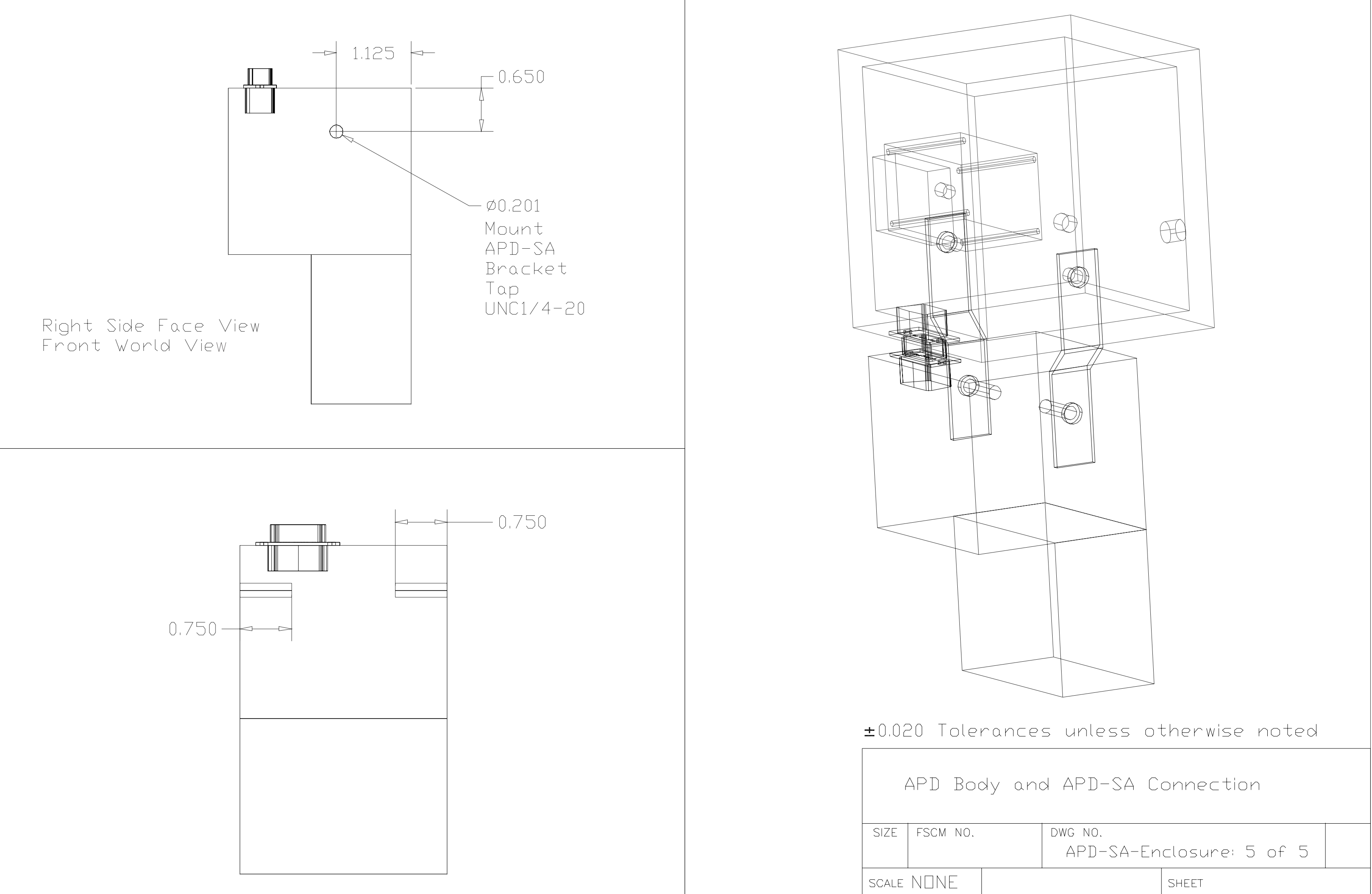
APD-SA Rendering and Assembly Notes			
SIZE	FSCM NO.	DWG NO. APD-SA-Enclosure: 1 of 5	
SCALE NONE		SHEET	


ø0.125
USB Mount
4 Holes
11/64 or 3/16
Loose Clearance
for UNC6-32
ø0.120
DB9-M Mount
2 Holes
9/64 or 1/8
Clearance for
UNC4-40
Bottom World View
Bottom Face Dimensions
ø0.125
DSP Mount
3 Holes
Clearance for
UNC4-40
ø0.313
DC Power
Connector
Right World View
Back Face
Dimensions
Right Side Face View
Front World View
Circuit Arrangement
in Enclosure
ADC and Logic
Circuit
DSP Circuit
USB Circuit
LCD Display
DB9-Male
Interface with
APD Body
±0.020 Tolerances unless otherwise noted except holes
See assembly notes for Bottom and Back tolerance issues
APD-SA Enclosure - Bottom, Back,
Circuit Arrangement
SIZE
FSCM NO.
DWG NO.
APD-SA-Enclosure: 2 of 5
SCALE NONE
SHEET


3.313
2.850
0.235
0.231
3.113
3.333
Top World View
Top Face Dimensions
Ø0.172
ADC and Logic Circuit Mount
11/64 or 3/16
Loose Clearance for
UNC6-32
Ø0.201
Note 1
(Left Face)
Enclosure Side Panel
Mount Bolt Holes
Bolt: 3.75 Length
under head
UNC1/4-20
Ø0.250
Clearance for
UNC1/4-20
(Right Face)
3.813
3.313
3.563 4.063
1.550
1.625
Note 1
(Drill #7 or
13/64
Tap UNC1/4-20)
Ø0.201
Note 1
Bracket Mount Holes
Front World View
Right Side Face Dimensions
3.833
3.333
1.406
1.413
2.460
2.484
4.063
3.563
Ø0.086
LCD Mount
4 Holes
3/32 or 7/64
Clearance for
M2.13-0.45
(Len-Pitch-mm)
0.519
0.629
0.852
2.482
2.719
2.814
Left World View
Face Dimensions
±0.020 Tolerances unless otherwise noted except holes
See assembly notes for Top and Front tolerance issues
APD-SA Enclosure - Top, Right Side,
Front Views
SIZE
FSCM NO.
DWG NO.
APD-SA-Enclosure: 3 of 5
SCALE NONE
SHEET



0.375
0.417
1.408
1.768
2.418
0.551
3.168
0.080
Ø0.300
Mount to APD-SA Enclosure
Use 1/2" Length under head UNC1/4-20
Ø0.300
Mount to APD Body
0.313
±0.05 Tolerances unless otherwise noted except holes
APD-SA to APD-Body Mounting Brackets
SIZE
FSCM NO.
DWG NO.
APD-SA-Enclosure: 4 of 5
SCALE NONE
SHEET


1.125
0.650
ø0.201
Mount
APD-SA
Bracket
Tap
UNC1/4-20
Right Side Face View
Front World View
0.750
0.750
±0.020 Tolerances unless otherwise noted
APD Body and APD-SA Connection
SIZE
FSCM NO.
DWG NO.
APD-SA-Enclosure: 5 of 5
SCALE NONE
SHEET

APPENDIX B: APD-SA PCB LAYOUT IMAGES

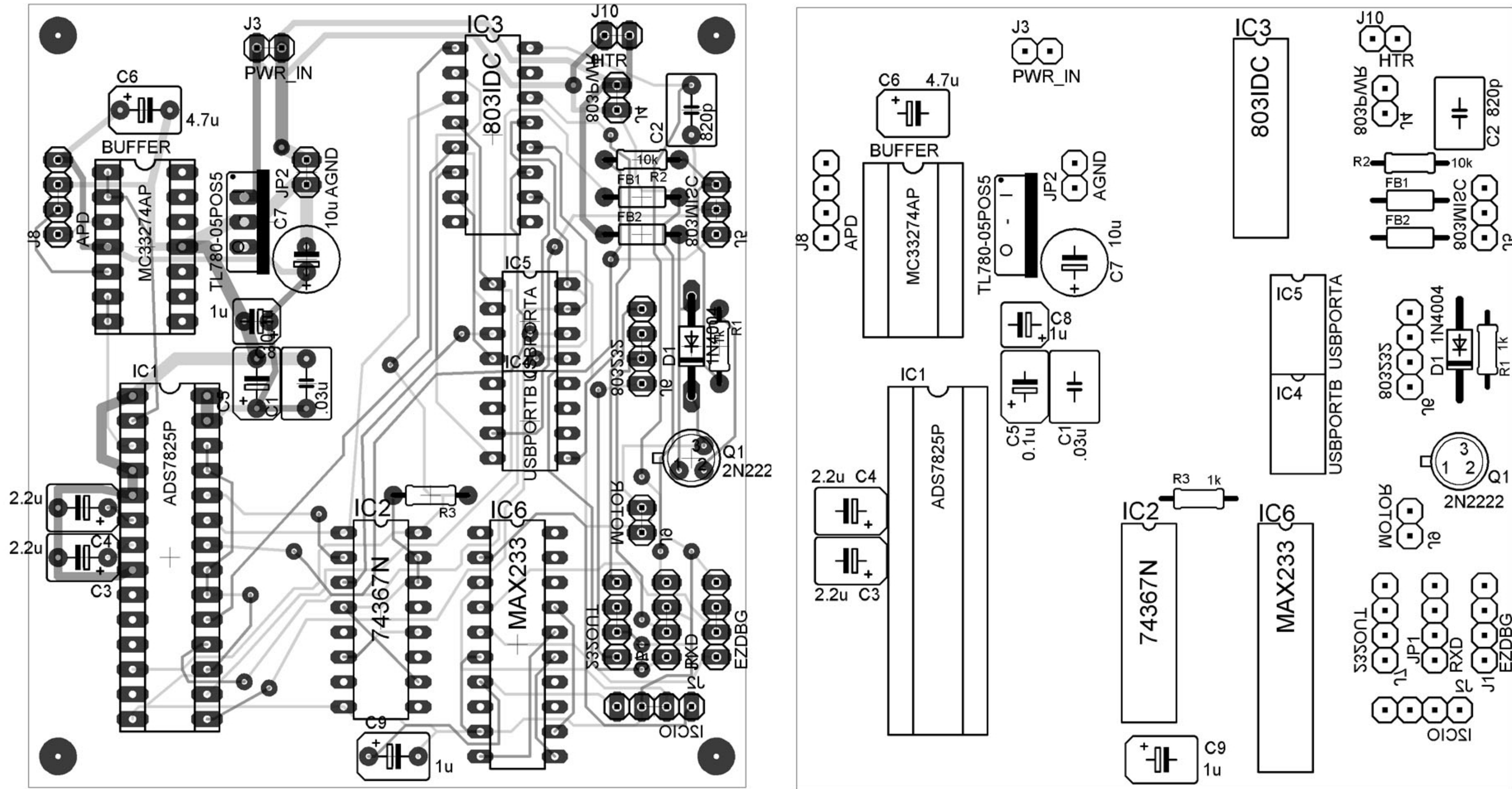


Figure B.1. APD-SA ADC and logic circuit PCB layout showing component placement from a top view

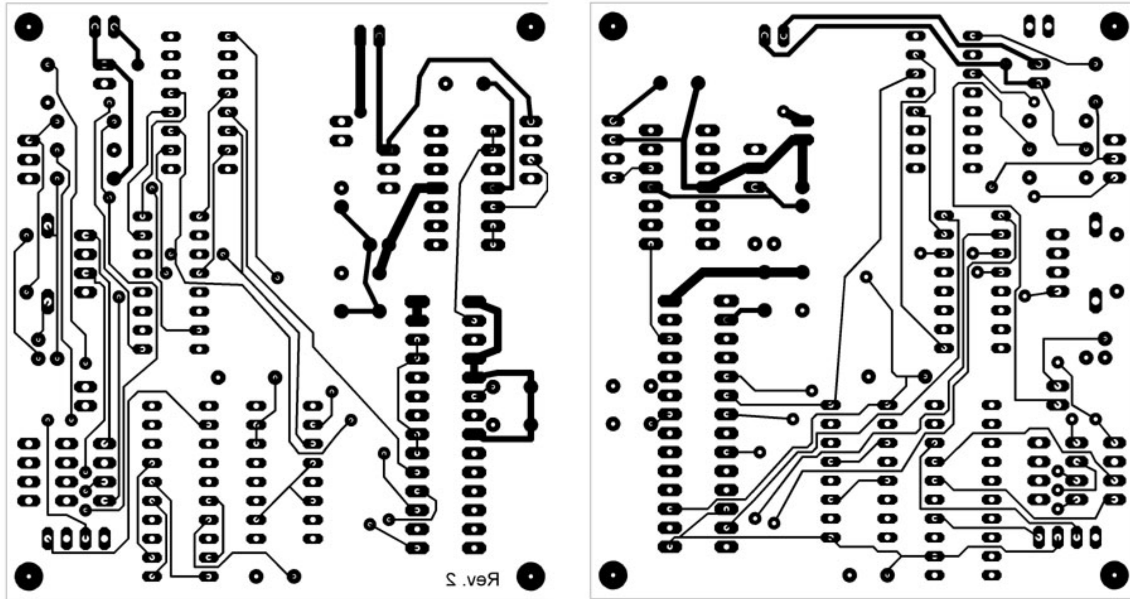


Figure B.2. APD-SA ADC and logic circuit PCB layout top (left) and bottom (right); top side is inverted so that when printed on laser jet transparency film, the ink side faces the PCB surface; should be scaled so that ICs are 0.3 inches between sides and 0.1 inches between neighboring pins

APPENDIX C: APD-SA PIN CONNECTIONS

Conventions:
- 803 refers to the Motorola DSP56F803 IC
- ADS refers to the ADS7825 ADC IC
- Mini refers to the New Micros NMIN-803 platform for the DSP56F803
- LCD0821 refers to the 2x8 character Matrix Orbital LCD screen
- J9 and similar refers to a header number on a PCB, numbered according to documentation for the platform
- Pin numbering corresponds to pin numbering in documentation for each chip or platform

Table C.1. Mini to ADS connections

ADS Pin	ADS Pin Function	803 Pin Function	Mini J9 Pin No.	803 Direction
15	DATACLK (in)	A0	7	out
16	SDATA (out)	A1	8	in
18	CHADDR MSB A1 (in)	A2	9	out
19	CHADDR LSB A0 (in)	A3	10	out
22	R/!C (in)	A4	11	out
24	!BUSY (out)	A5	12	in
23	!CS (in)	E4	6	out

Table C.2. APD DB9 connector; table orientation: facing the pins (not solder cups) on a male connector on the APD body

Flow	AGND (NC)	GND Sensor	V Heater	Motor+
1	2	3	4	5
6	7	8	9	
Pressure	V+ Sensor (7-35VDC)	V Heater	Motor-	

Table C.3. Mini to LCD0821; when board is oriented with pin headers at bottom and printed text on PCB reads upright, pin one is on the left

Pin	Function
1	RS232 Ground
2	RS232 RxD (Data in from 803 TxD)
3	Power (use 8-15 VDC in this wide voltage model)

Table C.4. Mini to USBI2CIO connections; USBI2CIO pin one of the IDC 34-pin header is top left-most with the single alignment slot facing left and USB jack on the right when looking at the top of the board

USBI2CIO Pin	USBI2CIO Pin Fcn	803 Pin Function	Mini J9 Pin	803 Direction
21	A0	A0	7	io
22	A1	A1	8	io
23	A2	A2	9	io
24	A3	A3	10	io
25	A4	A4	11	io
26	A5	A5	12	io
27	A6	A6	13	io
28	A7	A7	14	io
19	B6 (in) (INT6)	MOSI/RW/PE5	5	out
20	B7 (out)	MISO/RS/PE6	4	in
18	B5 (out)	IRQA (active low)	(Not J9)	in

Table C.5. 803 Mini J9 IDC to DIP via ribbon cable; IDC DIP is numbered as a standard DIP IC with pin one at the top left when the IC is oriented upright or vertically, as is typical in manufacturer documentation

Mini J9	IDC DIP	IDC DIP	Mini J9
16	8	9	15
14	7	10	13
12	6	11	11
10	5	12	9
8	4	13	7
6	3	14	5
4	2	15	3
2	1	16	1

APPENDIX D: DSP FIRMWARE CODE

Code Listings:

- Listing D.1. 56803_flash_xROM-RAM_linker.cmd – a linker command file for copying data ROM to RAM on startup)
- Listing D.2. 56F803RegDef.h – header file containing register declarations and macros for 56F803
- Listing D.3. M56803_main.c – main program code
- Listing D.4. 56803_vector_pROM.asm – interrupt vector table

```
# LISTING D.1. 56803_flash_xROM-RAM_linker.cmd

# ----------------------------------------------------
# Linker command file used for APD-SA USB with the New
# Micros NMIN-0803 module for the DSP56F803E chip.
#
# This linker command file is used to make the
# DSP copy data ROM to RAM on startup
# and makes provisions for storing constants
# defined in M56803_main.c as read-only
# in the FLASH memory as well.
#
# This linker file is based upon a memory map that
# uses only onboard memory for the DSP, i.e. no
# extended or external memory.
#
# This code is almost identical to sample code
# from Motorola as described below, but has been
# changed in a few places to facilitate
# the use of FLASH memory for calibration constant
# storage.
#
# N. Silverman 2004 for MS
#
# ----------------------------------------------
# Metrowerks, a company of Motorola
# sample code

# linker command file for DSP56803EVM
# using
#       flash pROM
#       flash xROM (const and dynamic data
#    internal xRAM (dynamic data copied from xROM)


# revision history
# 011020 R4.1 a.h. first version
# 030220 R5.1 a.h. improved comments

# ----------------------------------------------------



# see end of file for additional notes
# additional reference: Motorola docs
#   DSP56F801-7UM.pdf
#   DSP56F803EVMUM.pdf



# memory use for this LCF:
# interrupt vectors --> flash pROM starting at zero
#      program code --> flash pROM
#         constants --> flash xROM
#      dynamic data --> flash xROM (copied to RAM with init)



# requirements: Mode 0A and EX=0
```

```
# note -- there is a mode OB but any Reset or COP Reset
#         resets the memory map back to Mode 0A.



# DSP56803EVM eval board settings:
#    ON --> jumper JG4 (mode 0 upon exit from reset)
#   OFF --> jumper JG5 (enable external board SRAM)

# note: with this LCF memory config, internal memory
#       will be used regardless of JG5 setting;
#       however, you can turn SRAM off to validate
#       use of internal RAM.



# CodeWarrior debugger Target option settings
#    ON --> "Use Hardware Breakpoints"
#   OFF --> "Debugger sets OMR at Launch" option

# note: since debugger doesn't set OMR, JG4 on the EVM set OMR to:
# OMR:
#     0 --> MA bit
#     0 --> MB bit




# 56803
# mode 0A
# EX = 0

MEMORY
{
  .p_boot_flash_1       (RX)  : ORIGIN = 0x0000, LENGTH = 0x0004
  .p_interrupts_ROM     (RX)  : ORIGIN = 0x0004, LENGTH = 0x007C
  .p_flash_ROM          (RX)  : ORIGIN = 0x0080, LENGTH = 0x7D80
  .p_internal_RAM       (RWX) : ORIGIN = 0x7E00, LENGTH = 0x0200
  .p_boot_flash_2       (RX)  : ORIGIN = 0x8000, LENGTH = 0x0800
  .p_reserved           (RX)  : ORIGIN = 0x8800, LENGTH = 0x7800
  .x_compiler_regs_iRAM (RW)  : ORIGIN = 0x0030, LENGTH = 0x0010
  .x_internal_RAM       (RW)  : ORIGIN = 0x0040, LENGTH = 0x07C0
  .x_reserved           (R)   : ORIGIN = 0x0800, LENGTH = 0x0400
  .x_peripherals        (RW)  : ORIGIN = 0x0C00, LENGTH = 0x0400
  .x_flash_ROM          (R)   : ORIGIN = 0x1000, LENGTH = 0x1000
  .x_external_RAM       (RW)  : ORIGIN = 0x2000, LENGTH = 0xDF80
  .x_core_regs          (RW)  : ORIGIN = 0xFF80, LENGTH = 0x0080
}




# we ensure the interrupt vector sections are not deadstripped here

KEEP_SECTION{ interrupt_vectors.text, interrupt_vectors_mirror.text }
```

```
# the following is the section title for read-only data for FLASH
storage
# provision

KEEP_SECTION{ .rodata }


# About ROM-to-RAM copying at init time:

# In embedded programming,it is common for a portion of
# a program resident in ROM to be copied into RAM at runtime.
# For starters,program variables cannot be accessed until
# they are copied to RAM.

# To indicate data or code that is meant to be copied
# from ROM to RAM,the data or code is given two addresses.

# One address is its resident location in ROM (defined by
# the linker command file).The other is its intended
# location in RAM (defined in C code where we will
# do the actual copying).

# To create a section with the resident location in ROM
# and an intended location in RAM,you define the two addresses
# in the linker command file.

# Use the MEMORY segment to specify the intended RAM location,
# and the AT(address)parameter to specify the resident ROM address.


# we have defined the MEMORY segment x_internal_RAM above
# we set the .data section with AT to resident ROM address


SECTIONS
{

      .interrupt_vectors :
      {
          # from 56803_vector_pROM.asm
          * (interrupt_vectors.text)

      } > .p_interrupts_ROM



     # hawk mirrors this code back to P memory boot_flash_1

      .interrupt_vectors_mirror :
      {
          # from 56803_vector_pROM.asm
          * (interrupt_vectors_mirror.text)

      } > .p_boot_flash_2



      .executing_code :
      {
```

```
            # .text sections

            * (.text)
            * (rtlib.text)
            * (fp_engine.text)
            * (user.text)
      } > .p_flash_ROM



      .x_flash_ROM_data :
      {
          # initialized constants stay in flash xROM
            * (.rodata)

            # save address for ROM data we will copy to RAM
            __xROM_data_start =.;

      } > .x_flash_ROM





# the LCF command AT sets optional parameter
# that specifies the  address of the section.
# The default (if not specified)is to make
# the load address the same as the relocation address.
# Here, we want the load address to be in ROM
# and the relocation address in RAM.


      .data : AT(__xROM_data_start) # load address is in ROM
      {                             # starting after constant data
          # data sections

          # values inside this section represent relocated data
          __xRAM_data_start =.;

            # note:
            # for troubleshooting or other uses, you can directly
            # write data using the LCF and then use the memory window
            # to check results

            # WRITEH 0x0050;


         * (.data)
          * (fp_state.data)
            * (rtlib.data)

            __xRAM_data_end = .;
            __data_size = __xRAM_data_end - __xRAM_data_start;



            # .bss sections

            * (rtlib.bss.lo)
```

```
        __bss_start = .;

        * (.bss)

        __bss_end   = .;
        __bss_size = __bss_end - __bss_start;



        # setup the heap address

        __heap_addr = .;
        __heap_size = 0x0100;
        __heap_end = __heap_addr + __heap_size;

        . = __heap_end;



        # setup the stack address

        _min_stack_size = 0x0200;
        __stack_addr = __heap_end;
        __stack_end  = __stack_addr + _min_stack_size;
        . = __stack_end;



        # set global vars now

        # MSL uses these globals:
        F_heap_addr  = __heap_addr;
        F_heap_end   = __heap_end;
        F_stack_addr = __stack_addr;


        # stationery init code uses these globals:
        F_data_size     = __data_size;
        F_data_RAM_addr = __xRAM_data_start;
        F_data_ROM_addr = __xROM_data_start;
        F_bss_size      = __bss_size;
        F_bss_addr      = __bss_start;

     F_rom_to_ram    =  0x0001; # non-zero is true


  } > .x_internal_RAM   # relocation address --
                        # intended final destination in RAM


}



# ---------------------------------------------------------
# additional notes:
```

```
# about the reserved sections
# for this internal RAM and flash ROM LCF:

# p_interrupts_ROM -- reserved in pROM
# memory space reserved for interrupt vectors
# interrupt vectors must start at address zero
# interrupt vector space size is 0x80

# x_compiler_regs_iRAM -- reserved in internal xRAM
# The compiler uses page 0 address locations 0x30-0x40
# as register variables. See the Target manual for more info.

# other memory sections are per chip




# notes:
# program memory (p memory)
# (RWX) read/write/execute for pRAM
# (RX) read/execute for flashed pROM

# data memory (X memory)
# (RW) read/write for xRAM
# (R)  read for data flashed xROM

# LENGTH = next start address - previous
# LENGTH = 0x0000 means use all remaing memory
```

```
//LISTING D.2. 56F803RegDef.h

/********************************

 DSP56F803 Registers and related macro definitions
 including registers for APD-SA USB functionality

 An indented comment following a definition refers to the macro
 above the comment.  Non-indented comments refer to section
 below the comment.

 Register names that are not commented
 are consistent with DSP56F803 documentation.
 Therefore explanations for these macros can be found in
 documentation.  See:
   56F801-7UM.pdf (Users Manual)
   56800FM.pdf    (Family Manual)
   56F803.pdf     (56F803 Manual)

 Portions of GPIO defs were adapted from Small C
 examples from Peter Gray
 (http://petegray.newmicros.com)

 For a complete list of register definitions
 see 56803.h provided by Motorola in the development
 package for 56800 series

 N. Silverman 2004 for MS

*********************************/


//** General purpose IO (GPIOA) port A
#define GPIOA_BASE    0x0FB0
#define GPIOB_BASE    0x0FC0
#define GPIOD_BASE    0x0FE0
#define GPIOE_BASE    0x0FF0


//** General purpose IO (GPIOE) port E
#define GPIO_E_PUR    GPIOE_BASE+0x0
     //Pull-up enable register
#define GPIO_E_DR     GPIOE_BASE+0x1
     //Data register
#define GPIO_E_DDR    GPIOE_BASE+0x2
     //Data direction register
#define GPIO_E_PER    GPIOE_BASE+0x3
     //Peripherial Enable Register
#define GPIO_E_IAR    GPIOE_BASE+0x4
     //Interrupt assert register
#define GPIO_E_IENR   GPIOE_BASE+0x5
     //Interrupt enable register
#define GPIO_E_IPOLR  GPIOE_BASE+0x6
     //Interrupt polarity register
#define GPIO_E_IPR    GPIOE_BASE+0x7
     //Interrupt pending register
#define GPIO_E_IESR   GPIOE_BASE+0x8
     //Interrupt edge-sensitive register
```

```
//** ADC register definitions
//**    ADC Registers...
#define ADCR1           (*(char *) 0x0E80)
#define ADSDIS          (*(char *) 0x0E85)
#define ADSTAT          (*(char *) 0x0E86)
#define ADRSLT0         (*(char *) 0x0E89)
#define ADRSLT4       (*(char *) 0x0e8d)
#define ADLST1        (*(char *) 0x0e83)
#define ADLST2        (*(char *) 0x0e84)


//**    ADC Commands...
#define STARTADC 0x2005
     // Applies to ADCR1
#define ENSMPL0       254
     // 0x007e
    // Applies to ADSDIS - disables
     // all but sample 0 in seq mode
#define TESTADC       0x4000
     // Applies to ADSDIS - sets input of AN to Vref/2
#define CONVIP        0x8000
    // Applies to ADSTAT - conversion in
     //progress = bit 15 = 1
#define CH0RDY        0x0001
     // Applies to ADSTAT - channel 0 ready for
     // read is indic. by bit 0 = 1


// GPIO registers - port E
#define PEPUR   (*(char *) 0x0FF0)
#define PEDR    (*(char *) 0x0FF1)
#define PEDDR   (*(char *) 0x0FF2)
#define PEPER   (*(char *) 0x0FF3)
#define PEIAR   (*(char *) 0x0FF4)
#define PEIENR  (*(char *) 0x0FF5)
#define PEIPOLR (*(char *) 0x0FF6)
#define PEIPR   (*(char *) 0x0FF7)
#define PEIESR  (*(char *) 0x0FF8)


// GPIO register - port A
#define PAPUR   (*(char *) 0x0fb0)
#define PADR    (*(char *) 0x0fb1)
#define PADDR   (*(char *) 0x0fb2)
#define PAPER   (*(char *) 0x0fb3)
#define PAIAR   (*(char *) 0x0fb4)
#define PAIENR  (*(char *) 0x0fb5)
#define PAIPOLR (*(char *) 0x0fb6)
#define PAIPR   (*(char *) 0x0fb7)
#define PAIESR  (*(char *) 0x0fb8)


// SCI registers - RS232 communication
#define SCI0BR  (*(char *) 0x0F00)
#define SCI0CR  (*(char *) 0x0F01)
#define SCI0SR  (*(char *) 0x0F02)
#define SCI0DR  0x0F03
```

```
// PWM Registers
#define PWMA_BASE          (*(char *) 0x0e00)
#define PWMA_CTL      (*(char *) 0x0e00)
#define PWMA_OUT      (*(char *) 0x0e03)
#define PWMA_CNT      (*(char *) 0x0e04)
#define PWMA_MCM      (*(char *) 0x0e05)
#define PWMA_VAL0          (*(char *) 0x0e06)
#define PWMA_PMDISMAP1     (*(char *) 0x0e0d)
#define PWMA_PMDISMAP2     (*(char *) 0x0e0e)
#define PWMA_CFG      (*(char *) 0x0e0f)
#define PWMA_PMFSA    (*(char *) 0x0e02)


// Timer A0,B0, D0, D1, D2 register definitions
// not all are used in APD-SA USB code final revision
// but are retained for future development if needed
// Timer D is used extensively in final revision and thus
// has many more macros
#define TMRA_BASE     0x0d00
#define TMRA0_CTRL    (*(char *) 0x0d06)
#define TMRA0_CMP1    (*(char *) 0x0d00)
#define TMRA0_CMP2    (*(char *) 0x0d01)
#define TMRA0_CNTR    (*(char *) 0x0d05)
#define TMRA0_LOAD    (*(char *) 0x0d03)
#define TMRA0_SCR          (*(char *) 0x0d07)

#define TMRB_BASE     0x0d20
#define TMRB0_CTRL    (*(char *) 0x0d26)
#define TMRB0_CMP1    (*(char *) 0x0d20)
#define TMRB0_CMP2    (*(char *) 0x0d21)
#define TMRB0_CNTR    (*(char *) 0x0d25)
#define TMRB0_LOAD    (*(char *) 0x0d23)
#define TMRB0_SCR     (*(char *) 0x0d27)

#define TMRD0_SCR_R   0x0d67
     //TMRD_BASE + 0x7
#define TMRB0_SCR_R   0x0d27

#define TMRD2_BASE    0x0d20
#define TMRD2_CMP1    (*(char *) 0x0d30)
#define TMRD2_CMP2    (*(char *) 0x0d31)
#define TMRD2_CAP          (*(char *) 0x0d32)
#define TMRD2_LOAD    (*(char *) 0x0d33)
#define TMRD2_HOLD    (*(char *) 0x0d34)
#define TMRD2_CNTR    (*(char *) 0x0d35)
#define TMRD2_CTRL    (*(char *) 0x0d36)
#define TMRD2_SCR          (*(char *) 0x0d37)

#define TCSR          (*(char *) 0x0e7c)
#define TMODE         0x0080             //use TMODE
#define TISR1         (*(char *) 0x0e79) //includes 30,31
#define TISR2         (*(char *) 0x0e7a) //includes 42
#define T42           0x0400
#define T31           0x8000


//** on 56F803 Timer D is dedicated quad timer - reg defs
#define TMRD_BASE          0x0d60
```

```
#define TMRD0_CTRL         (*(char *) 0x0d66)
     //TMRD_BASE+0x6
#define TMRD1_CTRL         (*(char *) 0x0d6e)
     //TMRD_BASE+0xe
#define TMRD0_CNTR         (*(char *) 0x0d65)
     //TMRD_BASE+0x5
#define CRE                0xe000
     //0x2000 count rising edges
#define IPBD128            0x1e00
     //0x1e00 source = IPBus (40MHx for 8MHz PLL)
     //divided by 128  this is 312.5kHz thus at count
     //625, this is one 500 Hz cycle
#define CTR1OUT       0x0a00
     //source=counter 1 output
#define COUNTREP 0x0000
     // bit 6 = 0 count repeatedly
#define REINIT        0x0020
     // bit 5 = 1 = reinitialize on reach CMP1 value
     // for CMP1 count up compare
#define COUNTUP       0x0010
     // bit 4 = count up
#define SETOFLAG 0x0000
     // bits 2-0 = set OFLAG on successful compare

//** Timer D setup values - see documentation for
//** extensive description
//** of registers and meaning
#define SETUPTMRD0    CRE|CTR1OUT|REINIT|COUNTUP
#define SETUPTMRD1    0x3e30
#define TMRD0_SCR          (*(char *) 0x0d67)
     //TMRD_BASE+0x7
     // Timer D 0 status and control register
#define TMRD1_SCR          (*(char *) 0x0d6f)
#define TCF           0x8000
     // this is Timer Compare Flag bit
     // it needs to be cleared when set at reach compare
#define TCFIE         0x4000
     // this is bit 14 timer compare flag interrupt enable
#define TMRD0_CMP1    (*(char *) 0x0d60)
     // TMRD_BASE+0x0
     // Timer D compare register CMP1 for up counting compare
#define TMRD1_CMP1    (*(char *) 0x0d70)
#define TMRD1_CMP2    (*(char *) 0x0d71)
#define TMRD1_CNTR    (*(char *) 0x0d6d)
#define CMP1VAL       0xffff
     // 625 decimal - this is the value to which TMRD0
     //is compared
#define TMRD0_CMP2    (*(char *) 0x0d61)
     // TMRD_BASE+0x1
#define CMP2VAL       0x0000
#define TMRD0_LOAD    (*(char *) 0x0d63)
     //TMRD_BASE+0x3
     // Timer D 0 load value - should be 0x0000 to start from 0
#define TMRD1_LOAD    (*(char *) 0x0d6b)
#define TMRINITVAL    625
     // 0 to begin with

//** Timer D Chan 0 has interrupt vector 30  at 0x003C
//** IRQ table address
```

```
//** since it is vector 30 it has group priority register
//** 7 (along with vectors 28 through 31)
#define ITCN_BASE          0x0E60
     // from table 3-11 in DSP56F801-7UM.pdf
#define ITCN_GPR7          (*(char *) 0x0e67)
     // ITCN_BASE=0x0e60+$7 TMRD0
#define ITCN_GPR10    (*(char *) 0x0e6a)
     // TMRA0 = Ch 42 = 0x0700
#define ITCN_GPR9          (*(char *) 0x0e69)
     // TMRB0
#define TMRA0_IPL     0x0100
#define TMRB0_IPL          0x0100
#define TMRD2_IPL          0x0001
#define TMRD0_IPL          0x0100
     //0x1000 ch31 for TMRD1
     //0x0700 Ch 30 for TMRD0
#define TMRD1_IPL     0x1000
     // this puts 7 (highest priority) in the vector bits in
     // the grp 7 reg
     // since the 3rd nibble is for vector 30 in grp 7 prior
     // lev reg

//** Interrupt priority register
#define IPR_W         (*(char *) 0xfffb)
     // address of IPR
#define IPRVAL        0xFE12
#define IPRVAL_NOIRQ  0xfe00
#define ALLOWIRQA_FES 0x0006
     // allow IRQ A falling-edge sensitive
#define ALLOWIRQA_LLS      0x0002
     // allow IRQ A low-level sensitive
#define ALLOWIRQB_FES   0x0030
     // value written to it: enables all interrupt channels for
     // on-chip peripherals, but not ext IRQs


//** Flash programming registers
#define DFIU_CNTL   (*(char *) 0x0f60)
#define DFIU_EE     (*(char *) 0x0f62)
#define DFIU_PE     (*(char *) 0x0f61)
#define DFIU_IE     (*(char *) 0x0f65)
```

```
//LISTING D.3.  M56803_main.c

/*******************************************************

  m56800_main.c

  Main program code for APD-SA USB device
  for DSP56F803 firmware on NMIN-0803 from New Micros

  See MS Thesis for description of operation as well
  as diagram of program flow, use of interrupts, etc.

  Uses initialization code from Motorola embedded package.

  For additional DSP56F803 documentation see:
   56F801-7UM.pdf (Users Manual)
   56800FM.pdf    (Family Manual)
   56F803.pdf     (56F803 Manual)

  This code contains main program loop, interrupt functions,
  custom initialization code, APD perturbation detection
  algorithms, motor control code, LCD diplay code, USB
  communication code, ADC control code.

  An indented comment following a definition refers to the
  macro above the comment.  Non-indented comments refer to
  section below the comment.  This applies in macros, function
  and global variable declaration section.

  Some unused code remains from original
  prototype as a reference, and for potential
  future development.

  Serial communication code for character transmission
  was taken or adapted from examples from Pete Gray
  Small C examples (http://petegray.newmicros.com)

  N. Silverman 2004

*************************************************/

/**************************
*Includes
**************************/

//** Standard library
#include <stdlib.h>
//** Custom register macro definitions
#include "56F803RegDef.h"
//** ISRs are now contained within this code
//#include "isr.h"
//** Floating point support
#include <float.h>
#include "C:\Program Files\Motorola\Embedded
SDK\src\dsp56803evm\nos\include\port.h"
#include "C:\Program Files\Motorola\Embedded
SDK\src\dsp56803evm\nos\include\arch.h"
#include "C:\Program Files\Motorola\Embedded
SDK\src\dsp56803evm\nos\include\basic_op.h"
```

```
/***************************
*Pre-Macro Globals
***************************/

Frac16 pressval = 0.0f;
Frac16 flowval = 0.0f;


/*******************************
* Macros
*******************************/

#define SIZE               10
#define PI            3.14159f
     //PI needed for sim data only
#define SAMPFREQ           500
     //AD sample rate
#define MINPERTFREQ            10.0
     //6.0f
#define MAXPERTFREQ            16.0f
     //max considered pert freq
#define TIMERMS            2
     //multiplier * ms period of timer for soft
     //interrupt calls
#define MONITORTIME            1
     //monitor elapsed execution time?
//#define NUMVARS              2

#define PRESSCHAN              0
#define FLOWCHAN           1
#define SAMPBUFLEN             40
     //AD sample buffer length
#define RRVALLEN           20
     //number of rr measurements to average total
#define RRVALTILLDISP          3
     //num of rr meas at which to begin disp processing
#define MAXRR              20
     //maximum allowed rr value
#define REZEROSAMPLES          500
     //1 second at 500Hz

#define EQUALAVG           0
     //equally weighted average
#define LINEARAVG              1
     //linear wieghted average
#define NONLINAVG              2
     //exponentially weighted average
#define SUM           3
     //simple sum of values
#define MAXPROCESSES           1000
     //num of buffer process loops until quit
#define MINPERTLEN             14
     //SAMPFREQ / MAXPERTFREQ / 2.0
#define MAXPERTLEN             32
     //SAMPFREQ / MINPERTFREQ / 2.0
//#define TESTSWITCH           (int)(1.0*SAMPFREQ)
     //switch test display every second
```

```
//#define TESTSWITCHFINE            (int)(0.05*SAMPFREQ)
     //switch faster test display every 50 ms
//#define TESTSWITCHLONG            (int)(500.0*TESTSWITCH)

#define MAXRDALLOW             100.0
#define MINRDALLOW             0.0
     //maxs and mins for range of allowable values
     //min num samples required for TRUE pert
#define MINFLOW            180
     //200 is approx 0.25 Lps//50
     //this is now fraction value (int)//0.25
     //minimum allowed flow in LPS
#define MINPERT            40
     //minimum magnitude in counts of flow perturbation
#define OK           1
#define APDFAIL            -9
#define TOOMANYFAILS           5000
#define MINRD              0.2f
     //important criteria for ignoring zeroed rd values
     //MIN is inclusive
#define MAXRD              2.2f
     //MAX is not inclusive
#define NUMBINS            20
     //num bins for histogram of most common vals for rthres
#define NOMINALRTHRES          1.0f
     //default pert detection threshold
#define THRESPAD           interval
     //padding to increase resist thres for detection
     //"interval" sets it to bin interval in MCV detection
#define ERRPAD             0.051f
     //pad to add to close to borderline values for exactness
#define INH          1
#define EXH          -1
#define EXHISNEG           < 0
     //EXHISNEG > 0 for EXH = 1 not -1
#define TARGET_RESOLUTION           1
     //1-millisecond target resolution
#define DIGOUTPRECIS           2
     //num dec places to output to screen

#define CLOCKSPD           233
     //Assumed to be MHz
#define CYCLENS            40
     //Typical ns for one instr exec
#define CTRLMASK           0//^0x0b
     //these bits hardware inverted
#define STATMASK           0//^0x80
     //these bits hardware inverted
#define CTRLDIROUT             &(~0x20)
     //turn off direction bit 5 to output
#define DPDIG              3
     //dec pt req'd for digit 3 (base 1)

//** Mode Macros
#define NOCALLMODE             0x0000
#define STARTMODE              0x1000
#define CHANGEMODE             0x2000
#define STOPMODE           0x3000
#define ENDCALMODE             0x4000
```

```
#define NOCALLMODEECM                      0x5000
     //no call after ENDCALMODE called
#define POLLMODE           0x6000
#define ENDRUNMODE             0x7000
#define NOCALLMODEERM          0x8000
     //no call after end run mode
#define RUNMODE            0x9000
#define STREAMMODE             0xa000
#define USBRUNMODE             0xb000
#define POLLMASK           0xf000
     //mask for all of the above modes
#define USBATTACHMASK          0x0002
     //set if USB attach ok

#define ADS_DDR            0x001d
     //direction register for PA 0=input
#define DT_USB_DDR             0x00ff
     //direction register for PA 0=input
#define DF_USB_DDR             0x0000
     //direction register for PA...data from USB


//** USBI2CIO Macros
#define DR_to_USB              PEDR = PEDR |0x20
     //set PE5 to indicate data ready for USB to pick-up
#define NDR_to_USB             PEDR = PEDR &(~0x20)
     //clear PE5 to indicate data no data ready
#define USB_ACK            PEDR &0x40
     //read PE6 to see if USB has set ack bit
#define USB_DSP_ACK            PADR = 0xbb
     //set 0xbb on port
#define USB_HELLO              0xb0
     //USB device should send this on port to show it's there
#define USB_REQUEST_VARIABLES              0x02
     //send all relevent variables
#define USB_END_OF_VARIABLE_TRANSFER       0xfd
#define USB_BYTES_PER_VAR                  6
     //number of bytes for each value to send
#define USB_NUMVARS            36
     //number of total variables to send in variable request
#define USB_STREAM_ADC              0x03                          //
stream adc values
#define USB_STOP           0xfb
     //stop streaming stuff to USB
#define USB_WRITE_VARS              0x04
     //write variables to flash memory
#define USB_GET_HOST_VARS                  0x05
     //get vars from host
#define USB_RUN            0x06
     //run mode on APD with streaming of data
#define USB_DEC_PT             46
     //ASCII Decimal point
#define USB_NEG_SIGN           45
     //ASCII Hyphen
#define USB_MAX_NUM_TRIES_IN               5
     //maximum number of times to try after failing
     //to read a byte
#define USB_HOST_TOO_SLOW_STREAM           500
     //maximum timer ticks IP/64 until too slow during
```

```
     //streams of two channels of data
#define USB_HOST_TOO_SLOW           0xfa
     //host is too slow at picking up data



//** LCD Macros
#define CLS1    outscibyte(SCI0DR, 0xFE);
#define CLS2    outscibyte(SCI0DR, 0x58);
#define GOTOLINE2     outscibyte(SCI0DR, 0xFE);
                              outscibyte(SCI0DR, 0x47);
                              outscibyte(SCI0DR, 0x01);
                              outscibyte(SCI0DR, 0x02);

//** ADC Macros
//#define SETdataOUT  (short)_outp(control, 0x00 CTRLDIROUT CTRLMASK)
#define NADSBITS 16
     //7825 is 16bit; 7824 is 12bit
#define PADREADBITS   2
     //extra bits to read
#define RTSHIFTBITS   1
     //dummy trailing bits
#define LEADDUMMYBITMASK   0xf0000
     //bits to zero after right shift 0xf000 for
     //12 bit 0xf0000 for 16 bit
#define SETADSADDRCH0 PADR = PADR |0x08
     //(short)_outp(data, 0x00)
#define SETADSADDRCH1 PADR = PADR &(~0x0c)
     //(short)_outp(data, 0x10)
#define R_C_0    &(~0x10)
     //&(~0x04)
#define R_C_1    |0x10
     //|0x04
#define DCLK_0   &(~0x01)
     //&(~0x08)
#define DCLK_1   |0x01
     //|0x08
#define SELECTADS     PEDR = PEDR &(~0x10)
     //clear PE4 to select ADS
#define DESELECTADS   PEDR = PEDR |0x10
     //set PE4 to deselect ADS
//#define READADSSYNC (short)((_inp(status)STATMASK)&0x40)
#define READADSdata   PADR &0x02
     //(short)((_inp(status)STATMASK)&0x20)
#define READADSBUSY   PADR &0x20
     //(short)((_inp(status)STATMASK)&0x80)
#define NUMADBITLEV   65535.0
     //16bits is 65535.0; 12 bits is 4095.0
#define ADMIN    -10.0
#define ADMAX    10.0
#define ADCAL    (ADMAX/NUMADBITLEV*2.0)
     //in volts per bit levels for two's complement conversion
#define NUMSECCALAVG  5
     //number of seconds to average for calibration
#define MINVOLTAGE    0.25
     //minimum expected sensor voltage
#define MAXVOLTAGE    4.25
     //maximum expected sensor voltage
```

```
//**Motor Macros
#define MOTOR_0 | 0x02
     //MOTOR off
#define MOTOR_1 &(~0x02)
     //motor is 0x02 of control PORT _ !!!reversed
     //logic due to port functioning
#define MOTOR_ON (short)_outp(control, 0x00 CTRLDIROUT MOTOR_1
CTRLMASK);
#define MOTOR_OFF     (short)_outp(control, 0x00 CTRLDIROUT MOTOR_0
CTRLMASK);
#define PWM_PD_STEP   0x000f
     //increment/ decrement in PWM period
#define PWM_PD_INIT   0x0113
     //initial PWM period
#define PWM_PD_MIN    0x00c8
     //minimum allowed PWM pd during run

#define DEFPRESSOFF   7400 //2046//1.9283
     //default pressure offset voltage
//#define DEFPRESSSPAN     -35.588
     //default pressure span
#define DEFPRESSSPAN  -.003878
     //cmH20 per count  //-17.794
//#define DEFPRESSSPAN     -8.897
#define DEFFLOWOFF    7400 //2043//1.5630
     //default flow offset voltage
#define DEFFLOWSPAN   -.001252
      //Lps per count
      //-5.75
      //default flow span (Lps/V)
//#define DEFFLOWSPAN -2.875
#define PRESSCALVAL   5.0
     //press calibration value in cmH20
#define FLOWCALVAL    3.0
     //flow calibration value this is volume

#define STARTSTOPPORT status
     //use status port for push-button
#define STARTSTOPPORTMASK  STATMASK
     //use status port mask for TRUE input
#define STARTSTOPBIT  0x10
     //use bit 4 (base 0) as push-button bit
#define STARTSTOPHIT  0
     //assume normally pulled high and goes low with button

#define RUNBIT   0x00001
     //RUN MODE bits
#define TESTBIT  0x00010
     //
#define TESTSTREAMBIT 0x00020
     //
#define TESTSTREAMCHANBIT  0x00040
     //
#define CALBIT   0x00100
     //CAL MODE bits
#define CALVERIFYBIT  0x00080
     //determine verifybits
#define CALREZEROBIT  0x00200
     //
```

```
#define CALCHAN0BIT   0x00400
     //
#define CALCHAN1BIT   0x00800
     //
#define CALMODEBITS   0x00e80
     //determine REZERO, or CALCHAN bits
#define STOPBIT 0x01000
     //STOP MODE bit
#define NEWTIMERBIT   0x02000
     //if timer just reset and called first time
#define MOTORBIT 0x04000
     //motor on/ off bit on=1
#define CALLMODEBITS  0xf0000
     //CALLMODE (STOPMODE, CHANGEMODE etc) bit mask - 3 bits
#define FAILCODEBITS  0xf0000000
     //APDFAIL time requirement bit
#define FAILTIMEBIT   0x10000000

//assume each cycle about 40ns thus 1e9 * 40
#define DEBOUNCECNT   0x2f
#define MAXSHORTPULSECNT   0x3ff
#define MAXTESTCNT    0x8fff

#define REZERO_CMP    0xffff
     //at IP / 128 this is about 0.209 sec
#define SAMPLE500_CMP 0x0271
     //this is 500Hz at IP / 128
     //0x138 is 1kHz
#define SAMPLE250_CMP 0x4e2
     //at IP/128 this is 250 Hz

//** Debug options, file writing
#define DEBUG                 0
     //don't use with LCD0821
#define USELCD                1
     //19200 for LCD0821
#define SHOWEACHRRVAL         0
     //if yes, output each value, if no only show
     //progress and end values
#define SHOWRRASCIIINDICATOR  5
     //show '-' and '_' for RR val detected
     //number indicates interval to show indicator
#define SHOWRRBACKLIGHTINDICATOR   0
     //toggle backlight to indicate RR vals
#define VERBOSE 0
#define GENERATEdata  0
#define GENERATETESTdata   0
#define LOGdata 1
#define PROFILE 0
#define TESTFAIL 0
#define TESTDISPLAYSEGMENTS    0
#define REZEROONSTARTUP    1
#define MAXNUMPERTS   100
     //max no. perts to collect in each direction;
     // 0 is continuous
#define CHECKTIME     1
     //check execution time
#define TALK     1
     //for outsci type functions
```

```
#define STARTCHECKTIME     TMRA0_CTRL = 0x3c00
     //this is IP/64
     //0x3c00 start timer A0 at IP/64
     //0x3000 = ip/1
     //0x3600 ip/8
#define STOPCHECKTIME TMRA0_CTRL = 0x0000
     //stop timer A0
#define USEWEIGHTEDAVERAGE 0
     //whether or not to calculate weighted average
     //for each new pert detected
#define SERVOPWM 0
     //monitor PWM, freq period and adjust motor speed
#define SENDAVRVFRATRR     1
     //send actual as well as virtual flow rates at
     //pert rather than
     //- virtual is sent in place of the actual pressure
     //value at the rr time

//** for filtering
#define NZEROS        10
#define NPOLES        10
//#define GAIN        2.310287053e+00
      //for 200Hz, 4-pole butterworth
//#define GAIN    21.46710182352036
      //<- as 1/0.04658290663644
      //original: 2.146710182e+01
//#define GAIN   112.17732870747824
      //for 60 Hz
//#define GAIN   1240.14108757011457366
      //017187539115  //for 30 Hz
//#define GAIN   2.309503626e+04
      //30Hz 8pole Bessel
#define GAIN    7.856997806e+05
     //24Hz 10pole Bessel

//** RUNTIME changeable block locations
#define BLOCKcontrol  + 3
#define BLOCKMODE     + 0

#define FALSE    0
#define TRUE     1

#define C_DUTY   (*(char *) 0x1006)


/********************
* Function Prototypes
**********************/
void swap (int *a, int *b);
void print_array(int arr[], int length);
void inscichar(int *a, char *b);
void outsci(int *a, char *b);
void outscichar(int *a, char *b);
void outdec(int *chan, int val);
void outscihex(int *a, int b);
void outdecfloat (int *chan, float floatin);
void outscibyte (int *a, const int b);
int  ByteToUSB(unsigned int* c);
void outusb(unsigned int * val);
```

```
int  outusbdec(int var);
int  outusbdecfloat(float * var);
int  SendVars(void);
int  SendAckToUSB(void);

void SetupTimerInterrupt(const int comparevalue);
void SetupPWM(void);
void SetPWMDuty(int* duty);
void SetupADC();

void TimeProcRunMode();
void TimeProcStreamMode();

void GetADSSamples(short* first, short* second);
void InitConv(const short NextChan);
void LoopSWait(const long numcycles);
void LoopWait(const long millisec);

void isrIRQB ();
void isrTimerB0Compare ();

void ReadFlashConstants();
void WriteFlashConstants();

void DifferentiateArray(float* inarray,
      float* outarray, const int arraysize);
int  FullPertPresent(float* inarray, float* dinarray,
      const int arraysize, int * start, int * stop,
      float * thres);
int  FindMostCommonValue(float* inarray,
      const int arraysize,
      const float min, const float max, const int numbins,
      float * mcv);
int* MaxInt(int* inarray, const int arraysize);
int  ConsistentDirection(Frac16* inarray,
      const int arraysize, int * start, int * stop);
int  CalcRespResist(Frac16* pressdata, Frac16* flowdata,
      const int arraysize, int * start, int * stop,
      int * breathdir, float * rr);
int  OutputRespResistValue(float * rr);

short GetADSamples(short* first, short* second);
void ADSampleToVoltage(short* digital1, short* digital2,
      float* val1, float* val2);

void CopyCircArray(float* srcarr, float* destarr,
      float* bsrcarr, float* bdestarr, const int length);
void CopyCircArrayFrac(Frac16* srcarr, Frac16* destarr,
      Frac16* bsrcarr, Frac16* bdestarr, const int length);

int  ZeroOldPertRDPoints(float* curloc, float* begin,
      float* end, int * start, int * stop);
float WtAverage(float* inarray, int toteles2avg,
      const int type);

void ADVoltagesToScaledValues(Frac16* pressval,
      Frac16* flowval);

float* MaxFloat(float* inarray, const int arraysize);
```

```
static void DualFilterLoop4Pole100_500(Frac16* valinout1,
      Frac16* valinout2, const int reset);

float FracToFloat(int fraction);
Frac16 FloatToFrac(float floatin);

void SetupTimerCheck();
void StartProfileTimer();
void StopProfileTimer();

asm int GetMSByte (float floatin);
asm int GetLSByte (float floatin);

int ByteFromUSB(unsigned int* c);
int GetHostVars(void);
int inusbdecfloat(float* var);

void LCDGoToLineTwo(void);
void outscierr(char *b);
void LCDClearScreen(void);
void outsciusbinfo(char *b);



/******************************
* Post-Macro Globals
******************************/

//** Sample buffers
Frac16 pressbuf[SAMPBUFLEN] = {0.0f};
      //pressure values
Frac16 presstempbuf[SAMPBUFLEN] = {0.0f};
      //temp pressure values shifted
Frac16 flowbuf[SAMPBUFLEN] = {0.0f};
      //flow values
Frac16 flowtempbuf[SAMPBUFLEN] = {0.0f};
      //temp flow values shifted
float rdbuf[SAMPBUFLEN] = {0.0f};
      //device resistance values
float rdtempbuf[SAMPBUFLEN] = {0.0f};
      //device resistance values shifted
float drdbuf[SAMPBUFLEN] = {0.0f};
      //deriv of rd - shifted only
float freqbuf[SAMPBUFLEN] = {0.0f};
      //array of frequencies for motor control
float freqtempbuf[SAMPBUFLEN] = {0.0f};
      //shifted freqs for weighted averaging

//** Pointers to beginning of buffers
Frac16* press = &pressbuf[0];
Frac16* presstemp = &presstempbuf[0];
Frac16* flow = &flowbuf[0];
Frac16* flowtemp = &flowtempbuf[0];
float* rd = &rdbuf[0];
float* rdtemp = &rdtempbuf[0];
float* drd = &drdbuf[0];
float* freq = &freqbuf[0];
```

```
float* freqtemp = &freqtempbuf[0];
Frac16* pbpress = &pressbuf[0];//press;
Frac16* pbpresstemp = &presstempbuf[0];//presstemp;
Frac16* pbflow = &flowbuf[0];//flow;
Frac16* pbflowtemp = &flowtempbuf[0];//flowtemp;
float* pbrd = &rdbuf[0];//rd;
float* pbrdtemp = &rdtempbuf[0];//rdtemp;
float* pbdrd = &drdbuf[0];//drd;
float* pbfreq = &freqbuf[0];//freq;
float* pbfreqtemp = &freqtempbuf[0];//freqtemp;

//** Buffers for RR calculations
//** averages, shifting as in above buffers
float inrrvalsbuf[RRVALLEN] = {0.0f};
float inrrvalstempbuf[RRVALLEN] = {0.0f};
float exrrvalsbuf[RRVALLEN] = {0.0f};
float exrrvalstempbuf[RRVALLEN] = {0.0f};

//** Pointers to beginning of RR buffers
float* inrrvals = &inrrvalsbuf[0];
float* pbinrrvals = &inrrvalsbuf[0];//inrrvals;
float* inrrvalstemp = &inrrvalstempbuf[0];
float* pbinrrvalstemp = &inrrvalstempbuf[0];//inrrvalstemp;
float* exrrvals = &exrrvalsbuf[0];
float* pbexrrvals = &exrrvalsbuf[0];//exrrvals;
float* exrrvalstemp = &exrrvalstempbuf[0];
float* pbexrrvalstemp = &exrrvalstempbuf[0];//exrrvalstemp;

//** Calibration variables
Frac16 pressoff; //= DEFPRESSOFF;
Frac16 flowoff; //= DEFFLOWOFF;
float pressspan; //= DEFPRESSSPAN;
float flowspan; //= DEFFLOWSPAN;

//** Modes, counters
float rthres = 0.0f;
      //resistance threshold
long g_process = 0;
      //general counter for global monitors
int  timeprocdone = APDFAIL;
float displayvalue1 = 0.0f;
float displayvalue2 = 0.0f;
int  mode = 0x0000;
void  (*g_pfcn)();
      //pointer to whatever function is to be called in isr
float g_accumfloat1 = 0.0f;
float g_accumfloat2 = 0.0f;
int  duty; //= PWM_PD_INIT;

//** Calibration constant initialization for FLASH
//  Tables, structures or similar of stored constants
//  #pragma opt_dead_assignments off
//  #pragma opt_dead_code off
//** USE THIS (because it works):
#pragma use_rodata on
      //use this in combination with
      //KEEP SECTION { .rodata } in linker
      //to prevent dead-stripping of these
      //constants during compile/link
```

```
      //optimization
//#pragma define_section rodata "data.rodata" R
//#pragma section rodata begin

const Frac16 c_pressoff = DEFPRESSOFF;
      //these five lines should begin with const
const Frac16 c_flowoff = DEFFLOWOFF;
const float c_pressspan = DEFPRESSSPAN;
const float c_flowspan = DEFFLOWSPAN;
const int c_duty = PWM_PD_INIT;

//create dummy array to fill remainder of page
//to prevent linker from
//filling page with valuable info that gets erased
//when re-writing flash cal consts
const int fill[505] = {0};
      //512 bytes = 32 words = 8 rows per page of data flash

//#pragma section rodata end
//USE THIS (because it works):
#pragma use_rodata off
//#pragma opt_dead_assignments on
//#pragma opt_dead_code on
```

```
/*******************************************************
* MAIN PROGRAM LOOP AND CODE BEGINS HERE
*****************************************************/
int main(void)
{
     //int arr[SIZE] = {4,6,7,1,2,3,4,12,4,5};
     //int i,j;

     char c;
     int val;
     int  stat, result, ave, count;
     int  fint1, fint2, fdec1, fdec2;

     long longval;
     float floatval;

     //Disable interrupts to be enabled as needed
     IPR_W = 0x0000;

     if(TALK){ //config SCI output for RS232
          SCI0BR = 130; //130=19200
               //22; 22=11500 //65;
               //65 = 38400 baud
               //260
          //set up SCI (9600, 8N1
          SCI0CR = 0x0008;//12;
          //and display instructions...
          //config display - autoline wrap 254,67
          outscibyte(SCI0DR, 0xFE);
          outscibyte(SCI0DR, 0x43);

     }
     if(!TALK)
          SCI0CR = 0x0002;
            //set receiver to standby mode
            //and everything else disabled

     //Use fill data to keep in program
     stat=0;
     for(val=0;val<505; val++){
          stat *= fill[val];
          //outdec(SCI0DR, stat);
          outsci(SCI0DR, "APD v0.1");
     }

     //Clear screen
     outscibyte(SCI0DR, 0xFE);
     outscibyte(SCI0DR, 0x58);
     outsci(SCI0DR, "APD v0.1");

     //Block cursor off
     CLS1;
     outscibyte(SCI0DR, 0x54);

     CLS1; //prep for command;
     outscibyte(SCI0DR, 0x42);// 46 = off; 42 = backlight on
     outscibyte(SCI0DR, 0x05); //zero means inf minutes
```

```
//Read all constants from FLASH memory into
//global variables
ReadFlashConstants();

SetupPWM();

//** Set up GPIO E
PEIAR = 0;
PEIENR = 0;
PEIPOLR = 0;
PEIESR = 0;
PEPER = 0x0083;
     //let GPIO control bits 2,3,4,5,6
     //on the port - peripheral
     //is enabled only on bits 0,1,7
PEDDR = 0x003c;
       //bits 2,3,4,5 set to output
       //bit 6 = input
PEPUR = 0x00ff;
       //pull-ups //1=pull-up enabled on any inputs
val = 0x0008 | 0x0004;
PEDR = val;
       // set both leds 'on'

//** Set up GPIO A
PAIAR = 0;
PAIENR = 0;
PAIPOLR = 0;
PAIESR = 0;
PAPER = 0x0000;
       //let GPIO control all pins on port A -
       //not the peripheral
PADDR = 0x001d;
       //0 = input, 1 = output
PAPUR = 0x00ff;
       //use pull-ups on all bits
       //(only used if it's output)
PADR = 0x0000;

//SetupADC();
//
//** INIT ADS7824/5 to proper channel acquire
SELECTADS;
PADR = PADR R_C_1;
PADR = PADR R_C_1;
SETADSADDRCH0;
PADR = PADR R_C_0;
PADR = PADR R_C_0;
PADR = PADR R_C_1;
DESELECTADS;

//No Data ready to USB yet
NDR_to_USB;

//If necessary, setup timer for profiling
if(CHECKTIME)
     SetupTimerCheck();
```

```
//Test DSP to USB byte transmit
/*c = 0xcf;
g_process = 0;
STARTCHECKTIME;
while(1)
{
      //STARTCHECKTIME;
      if((TMRA0_CNTR &0xefff) > 0x7800)
      {
           g_process++; //clear sign bit
           TMRA0_CNTR=0;
      }
      if (g_process > 125)
      {
           STOPCHECKTIME;
           TMRA0_CNTR=0;
           outsci(SCI0DR, "s");
           STARTCHECKTIME;
           ByteToUSB(&c);
           g_process = TMRA0_CNTR;
           STOPCHECKTIME;
           TMRA0_CNTR=0;
           outscibyte(SCI0DR, 0xFE);
           outscibyte(SCI0DR, 0x58);
           outdec(SCI0DR, g_process);
           g_process=0;
           STARTCHECKTIME;
      }
}*/

//Debug print old flash constants:
if(DEBUG){
outsci(SCI0DR, "OFC: ");
outdec(SCI0DR, pressoff);
outdec(SCI0DR, flowoff);
outdecfloat(SCI0DR, pressspan);
outdecfloat(SCI0DR, flowspan);}

//Rezero device
if(REZEROONSTARTUP)
{
      LCDClearScreen();
      if(DEBUG){
           outsci(SCI0DR, "REZ...");}
      if(USELCD){
           outsci(SCI0DR, "Self-");
           LCDGoToLineTwo();
           outsci(SCI0DR, "Testing");}

      g_process = 0;
       //set this to zero; it will increment until enough
      //samples have been collected for the zeroing

      //Turn off any prior running timer
TMRB0_CTRL = 0x0000;

//Reset pointer to proper interrupt function
g_pfcn = TimeProcStreamMode;
```

```
//Should already be off
//Turn off motor
//SetPWMDuty(0x0000);

      //Reconfig and set timer
      SetupTimerInterrupt(SAMPLE500_CMP);

      longval = (long)REZEROSAMPLES;
      while(1)
      {
           //outdec(SCI0DR, longval);
           if (_L_sub(g_process, longval) >= 0)
           {
                //outsci(SCI0DR, "bigger");
                break;
           }
      }

      //Stop timer
      TMRB0_CTRL = 0x0000;

      //floatval = (g_accumfloat1 / (float)REZEROSAMPLES);
      //outdecfloat(SCI0DR,floatval);
      //outdecfloat(SCI0DR, g_accumfloat1);

      //floatval = 11243.00;
      //outdecfloat(SCI0DR, floatval);
      //pressoff = (int)floatval;

      pressoff = FloatToFrac(g_accumfloat1 /
           (float)REZEROSAMPLES);
      flowoff = FloatToFrac(g_accumfloat2 /
           (float)REZEROSAMPLES);

      //May want to write these Frac16 values
      //into the ADC offset registers
      //in future

      if(DEBUG)
      {
           outsci(SCI0DR, "pfx: ");
           outdec(SCI0DR, pressoff);
           outsci(SCI0DR, ", ");
           outdec(SCI0DR, flowoff);
      }

      LCDClearScreen();

      if(DEBUG){
      outsci(SCI0DR, "Dun");}

} //if(REZEROONSTARTUP)

//Write new re-zeroed constants to flash memory
WriteFlashConstants();

//Read them back into variables
ReadFlashConstants();
```

```
//Debug print old flash constants:
if(DEBUG){
outsci(SCI0DR, "NFC: ");
outdec(SCI0DR, pressoff);
outdec(SCI0DR, flowoff);
outdecfloat(SCI0DR, pressspan);
outdecfloat(SCI0DR, flowspan);}

//outscibyte(SCI0DR, 0xFE);
//command
//outscibyte(SCI0DR, 0x58);
//clear screen and go to top left

//If initial mode on power on is to be runmode then:
mode = mode | RUNMODE;

if(USELCD){
     outsci(SCI0DR, "Reading");
     LCDGoToLineTwo();
     //CLS1; //prepare for command
     //outscibyte(SCI0DR, 0x48);
     //go to home position for progress indicators

     CLS1; //prepare for command
     outscibyte(SCI0DR, 0x44);
             //auto line wrap off so keep
             //progress indicators only at top
}

//Now enable IRQA interrupt with falling
//edge sensitivity to prevent mult trigs
IPR_W = IPR_W | ALLOWIRQB_FES; //ALLOWIRQA_FES;


//*********************************
//** Main polling loops for modes etc.
//** loop forever
while (1){
     //since there is no operating system,
     //there is nothing to return to
     //so just loop forever - e.g.
     //that's the simple OS - polling
     //with IRQ adjusted commands

    if((mode & POLLMASK) == RUNMODE)
    {

          //Turn off any prior running timer
          TMRB0_CTRL = 0x0000;

          //Clear all buffers here...

          g_process=0;
          //signals TimeProcRunMode to rezero static vars.
          //it is set to 1 after re-zeroing in the procedure

          //Reset pointer to proper interrupt function
          g_pfcn = TimeProcRunMode;
```

```
        //SetPWM duty for initial
        duty = C_DUTY;//c_duty;//PWM_PD_INIT;
        SetPWMDuty(&duty);

        //Reset mode to no call mode so this
        //is not called again

        mode = (mode & ~RUNMODE);
        //mode = NOCALLMODE;

        //Reconfig and set timer
        SetupTimerInterrupt(SAMPLE250_CMP);

}

if((mode & POLLMASK) == USBRUNMODE)
{

        //Turn off any prior running timer
        TMRB0_CTRL = 0x0000;

        g_process=0;
        //signals TimeProcRunMode to rezero static vars.
        //it is set to 1 after re-zeroing in the procedure

        //Reset pointer to proper interrupt function
        g_pfcn = TimeProcRunMode;

        //SetPWM duty for initial
        duty = C_DUTY;//c_duty;
        SetPWMDuty(&duty);

        //Reset mode to no call mode so
        //this is not called again
        mode = (mode & ~USBRUNMODE);
        //mode = NOCALLMODE;

        //Reconfig and set timer
        SetupTimerInterrupt(SAMPLE250_CMP);

}

if((mode & POLLMASK) == STREAMMODE)
{

        //Turn off any prior running timer
        TMRB0_CTRL = 0x0000;

        //Reset pointer to proper interrupt function
        g_pfcn = TimeProcStreamMode;

        //Turn off motor
        duty = 0x0000;
        SetPWMDuty(&duty);

        duty = C_DUTY;

        //Reset mode so no call is made again to this stuff
        mode = (mode & ~STREAMMODE);
```

```
                //mode = NOCALLMODE;

                //Reconfig and set timer
                SetupTimerInterrupt(SAMPLE500_CMP);

           }

           if((mode & POLLMASK) == STOPMODE)
           {
                //Turn off any running timer
                TMRB0_CTRL = 0x0000;

                //Turn off motor
                duty = 0x0000;
                SetPWMDuty(&duty);

                duty = C_DUTY;

                //Reset mode - turn off stopmode bits
                mode = (mode & ~STOPMODE);
                //mode = NOCALLMODE;

           }

           }    //while (c != 'q') //end of main polling while loop


      return(0);

} //main

/************************************************************/
void WriteFlashConstants()
{

       //Writes FLASH constants to FLASH

      int status = 0x0000;
      Frac16 fraction = 0x0000;
      float floatval = 0.0;

      //disable IRQ interrupts
      //IPR_W = (IPR_W & ~0x0010);// & ~0x0002); //disable IRQA
      //nope, that should be done in calling routine

      //First intelligent erase a block of flash
      //the APD program so far uses the first page of flash memory
      //This is data flash
      if((DFIU_CNTL &0x803f) == 0x0000)
      //check to make sure in flash read or standby mode
      {
           //set IFREN to 0 to use main memory, not information block
           DFIU_CNTL = DFIU_CNTL & ~0x0040;

           DFIU_IE = DFIU_IE | 0x0100;
           //set IE[8] to enable trev interrupt when done erasing

           //erase page: page # = address &= 0x7fff and then >>= 8
           //page 0 right now for address = 0x1000 --> 0x0010
```

```
DFIU_EE = DFIU_EE | 0x4010;
//EE[14] = IEE = intelligent erase enable
//EE[0:6] = page number = 0x0010

//write any value to an address in the page and
//it will start erase cycle
//on the 803, xROM (flash) is from 0x1000 - 0x1fff
(*(char *) 0x1000) = 0x0000;
//any value written to data flash
//start address
//read busy bit until it clears indicating end of erase
do
{
     status = DFIU_CNTL & 0x8000;
     //busy = DFIU_CNTL[15]
} while(status != 0x0000);

//clear CNTL and EE registers
DFIU_EE = 0x0000;
DFIU_CNTL = 0x0000;
DFIU_IE = 0x0000;

//busy bit has cleared, now write data to block.
//setup intelligent page write
//if(DFIU_CNTL &0x803f = 0x0000)
//may need to use something like this <--

//Test here!!! for erase page only
DFIU_IE = DFIU_IE | 0x0100;
//set IE[8] to enable trev interrupt when done erasing

DFIU_PE = DFIU_PE | 0x4080; //PE= program enable
//PE[14] = intelligent program enable
//PE[0:9] = row number - start at 0
//address &= 0x7fff >>= 5 --> 0x0050

//32 words in a row - each word is 16 bits
//begin writing to each address
//use order of constants in global declaration before
//main()
fraction = pressoff;
(*(char *) 0x1000) = fraction;
do status = (DFIU_CNTL & 0x8000); while(status != 0);
//loop until not busy

//writing the flash
fraction = flowoff;
(*(char *) 0x1001) = fraction;
do status = (DFIU_CNTL & 0x8000); while(status != 0);
//loop until not busy

//writing the flash
floatval = pressspan;
fraction = GetMSByte(floatval);
(*(char *) 0x1003) = (fraction);
do status = (DFIU_CNTL & 0x8000); while(status != 0);
//loop until not busy

//writing the flash
```

```
            floatval = pressspan;
            fraction = GetLSByte(floatval);
            //Doesn't work:
            //status = (((int)floatval >> 16) & 0xffff);

            (*(char *) 0x1002) = fraction;
            do status = (DFIU_CNTL & 0x8000); while(status != 0);
            //loop until not busy

            //writing the flash
            floatval = flowspan;
            fraction = GetMSByte(floatval);
            //Doesn't work
            //(*(char *) 0x1004) = ((int)floatval & 0xffff);

            (*(char *) 0x1005) = fraction;
            do status = (DFIU_CNTL & 0x8000); while(status != 0);
            //loop until not busy

            //writing the flash
            floatval = flowspan;
            fraction = GetLSByte(floatval);
            //Doesn't work //status = (((int)floatval >> 16) & 0xffff);
            (*(char *) 0x1004) = fraction;
            do status = (DFIU_CNTL & 0x8000); while(status != 0);
            //loop until not busy

            //writing the flash
            fraction = duty;
            (*(char *) 0x1006) = fraction;
            do status = (DFIU_CNTL & 0x8000); while(status != 0);
            //loop until not busy

            //writing the flash

            //when finished writing all data clear enable bits
            DFIU_PE = 0x0000; //clear program enable register
            DFIU_CNTL = 0x0000;
            DFIU_IE = 0x0000;
            //Test here <----!!!

            if(DEBUG)outsci(SCI0DR, "WFO ");

        } else
        {
            if(DEBUG)outsci(SCI0DR, "WFX");
            if(USELCD)outscierr("WFX");
        }

        //re-enable IRQ interrupts
        //this should be done in calling routine

} //WriteFlashConstants


/*******************************************************/
asm int GetMSByte ( float floatin)
{
    //ASM routine to get most significant byte
```

```
     //floatin is passed on a (a1, a0)
     move a1,y0
     rts

} //GetMSByte


/*****************************************************/
asm int GetLSByte ( float floatin )
{
     //ASM routine to get least significant byte
     move a0,y0
     rts
} //GetLSByte


/******************************************************/
void ReadFlashConstants()
{

     //Read FLASH constants from FLASH into vars

     //pressoff = 0x0000;

     pressoff = (*(char *) 0x1000); //this works - others do not
     //(*(&c_pressoff)); //(*(char *) 0x1000);//c_pressoff;
     pressspan = (*(float *) 0x1002); //c_pressspan;
     flowoff = (*(char *) 0x1001); //c_flowoff;
     flowspan = (*(float *) 0x1004); //c_flowspan;
     duty = (*(char *) 0x1006); //c_duty;


} //ReadFlashConstants


/******************************************************/
int ByteToUSB(unsigned int* c)
{

     //Send Byte to EZ-USB chip

     int value;
     int wdog=0;

     DESELECTADS;
     PADDR = DT_USB_DDR;
     PADR = *c;
     //DR_to_USB; //was commented 1
     //do value = USB_ACK; while(value == 0);
     //LoopSWait(40);
     do
     {
          NDR_to_USB; //was uncommented 1
          DR_to_USB;  //was uncommented 1
          //LoopSWait(4);
          //try without this for quickness,
          //may need to re-structure method
          value = USB_ACK;
          wdog++;
```

```
    } while(value==0 && wdog<0x7ff);

    //de-assert data ready to you mr usb
    NDR_to_USB;

    //return values according to failure or success
    if(wdog>=0x7ff)
    {
        if(DEBUG)outsci(SCI0DR, "BTU1x");
        if(USELCD)outsciusbinfo("14");
        return APDFAIL;
    }

    //if it was a watchdog ok, then finish ack cycle...
    if(wdog<0x7ff)
    {
        //reset watchdog
        wdog=0;

        //wait for USB to ack that data was picked up and has fin.
        //processing it mkay
        do {
            //LoopSWait(4);
            value = USB_ACK;
            wdog++;
        } while(value!=0 && wdog<0xfff);
        //make sure the ACK bit is dropped
    }

    //CLS1;
    //outdec(SCI0DR, wdog);

    //return values according to failure or success
    if(wdog>=0x7ff)
    {
        if(DEBUG)outsci(SCI0DR, "BTU2x");
        //if(USELCD)outscierr("BTU2x");
        if(USELCD)outsciusbinfo("11");
        return APDFAIL;
    } else
        return OK;


    //scenario 1 doesn't work...locks up in current method

}


/**********************************************************/
int ByteFromUSB(unsigned int* c)
{

     //Read a byte of data from the EZ-USB chip

    int value;
    int wdog=0;

    DESELECTADS;
    PADDR = DF_USB_DDR;
```

```
	//PADR = *c;
	//DR_to_USB; //was commented 1
	//do value = USB_ACK; while(value == 0);
	//LoopSWait(40);
	//Signal Host with an interrupt
	do
	{
		NDR_to_USB; //was uncommented 1
		DR_to_USB;  //was uncommented 1
		LoopSWait(4);
		//try without this for quickness,
		//may need to re-structure method
		value = USB_ACK; //read USB ACK bit
		wdog++;
	} while(value==0 && wdog<0x7ff);

	//de-assert data ready to you mr usb
	//NDR_to_USB;

	*c = PADR; //read the port from USB

	//Now say I got that data with NDR_to_USB
	NDR_to_USB;

	//if it was a watchdog ok, then finish ack cycle...
	if(wdog<0x7ff)
	{
		//reset watchdog
		wdog=0;

		//wait for USB to ack that data was picked up and has fin.
		//processing it mkay
		do {
			value = USB_ACK;
			wdog++;
		} while(value!=0 && wdog<0x7ff);
		//make sure the ACK bit is dropped
	}


	//return values according to failure or success
	if(wdog>=0x7ff)
	{
		if(DEBUG)outsci(SCI0DR, "BX");
		if(USELCD)outscierr("BX");
		return APDFAIL;
	} else {
		if(DEBUG)
		{
			outscihex(SCI0DR, *c);
		}
		return OK;
	}

} //ByteFromUSB

/**************************************************************/
int GetHostVars(void)
```

```
{

       //Retrieve variables from the USB PC host

      int success=0;
      int i = 0x0000;
      float floatval;
      unsigned int dummy;

      success = inusbdecfloat(&floatval);
      if(DEBUG)
      {
           outdecfloat(SCI0DR, floatval);
      }
      //pressoff = FloatToFrac(floatval);
      pressoff = (int)floatval;
      if(DEBUG)
      {
           outdec(SCI0DR, pressoff);
      }
      success = inusbdecfloat(&floatval);
      //flowoff = FloatToFrac(floatval);
      flowoff = (int)floatval;

      success = inusbdecfloat(&floatval);
      //at present, pressspan is transmitted as inverse to maximize
      //transmitted signif digits
      floatval = 1.0 / floatval; //invert to make true value
      pressspan = floatval;

      success = inusbdecfloat(&floatval);
      floatval = 1.0 / floatval; //again, invert to make value needed
      flowspan = floatval;

      success = inusbdecfloat(&floatval);
      //duty = FloatToFrac(floatval);
      duty = (int)floatval;
      if(DEBUG)
      {
           outdecfloat(SCI0DR, duty);
      }
      do
      {
           success = ByteFromUSB(&dummy);
           //this is placed here to cause one more
      } while(dummy!=USB_END_OF_VARIABLE_TRANSFER);
      //INT6 strobe to the EZUSB that triggers it to
      //read for its USB_END_OF_VARIABLE_TRANSFER
      //byte from host and thus go through its stop transfer paces.

end_of_gethostvars:
;

if(success==APDFAIL)
      return APDFAIL;
else
      return OK;
} //GetHostVars()
```

```
/*****************************************************/
int SendVars(void)
{

     //Send all relevant variables to host computer on connect

    int i=0x0000;
    int var;         //pointer to variable
    char temp[USB_BYTES_PER_VAR]={USB_END_OF_VARIABLE_TRANSFER};
    int value = 0x00;
    unsigned int j=0;
    int success=0;
    float floatval;

    /*for(i=0; i<=USB_NUMVARS; i++)
    {
     //is it after last variable sent
     if(i==USB_NUMVARS)
     {
          //var = &temp[0]; //point to beginning of array
          //outusb
          //DESELECTADS;
               //PADDR = DT_USB_DDR;
               //PADR = USB_END_OF_VARIABLE_TRANSFER;
               //DR_to_USB;
               //LoopSWait(4);
               //do
               //{
               //   NDR_to_USB;
               //   DR_to_USB;
               //   LoopSWait(4);
               //   value = USB_ACK;
               //} while(value == 0);
               //NDR_to_USB;
               //
               j=(unsigned int)0xfd;//USB_END_OF_VARIABLE_TRANSFER;
               success = ByteToUSB(&j);
          outsci(SCI0DR, "L.V");
          value = USB_END_OF_VARIABLE_TRANSFER;
          outscihex(SCI0DR, value);
          //outusb(&temp[0]);
          goto end_of_sendvars;
     }


     j=(unsigned int)i;

     outscihex(SCI0DR, i);
     //ByteToUSB(&j);

     /*var = (int)pressoff;         //set var to address of variable
     if(sizeof(var)==sizeof(Frac16))
     {
          outsci(SCI0DR, "SNDVAR");
          outdec(SCI0DR, var);
          outsci(SCI0DR, "_");

          g_process=0;
```

```
                TMRA0_CNTR=0;
                STARTCHECKTIME;

                success = outusbdec(var);

                g_process = TMRA0_CNTR;
                STOPCHECKTIME;
                TMRA0_CNTR=0;
                outscibyte(SCI0DR, 0xFE);
                outscibyte(SCI0DR, 0x58);
                outdec(SCI0DR, g_process);
                outsci(SCI0DR, "_");
                g_process=0;

          }
     //if(sizeof(*var)==sizeof(float)){outusbdecfloat((float *)var);}
     ////outusb(var);
     //send the var out of port -  outusb need to check type
     }*/

     var = (int)pressoff;
     success = outusbdec(var);

     var = (int)flowoff;
     success = outusbdec(var);

     floatval = 1.0 / pressspan;
     //invert to trasmit maximing signif digs trans.
     success = outusbdecfloat(&floatval);

     floatval = 1.0 / flowspan;
     //invert to trasmit maximing signif digs trans.
     success = outusbdecfloat(&floatval);

     var = (int)C_DUTY;//c_duty;
     //use c_duty because duty will have been set to
     //0x0000 on connect to dsp
     //via STOP_MODE
     success = outusbdec(var);
     duty = var; //this is done so that when WriteFlashVars
     //is called, it write the proper value
     //this may cause problems later, depending on how
     //duty updates are implemented
                    //NOTE NOTE WARNING CAUTION of above

     j=(unsigned int)0xfd;//USB_END_OF_VARIABLE_TRANSFER;
     success = ByteToUSB(&j);
     if(DEBUG)outsci(SCI0DR, "URe");


     //INT6 strobe to the EZUSB that triggers it to
     //read for its USB_END_OF_VARIABLE_TRANSFER
     //byte from host and thus go through its stop transfer paces.

end_of_sendvars:
;

if(success==APDFAIL)
     return APDFAIL;
```

```
else
     return OK;
}

/*******************************************************/
void outusb(unsigned int * val)
{

      //Send an integer value to the EZ-USB
      //maximum USB_BYTES_PER_VAR value

     int i = 0;

     //val is assumed to point to the first element in an array of
     //bytes to send
     for(i=0; i<USB_BYTES_PER_VAR; i++)
     {
          ByteToUSB(val);
          val++;
     }
}

/*******************************************************/
int outusbdec(int var)
{

     //this should work for Frac16 as well
     //send out digits of an int as chars via port A
     //assumes that values are not large - need support for positive
     //and negative integers
     //also note proper number of digits
     unsigned int u,t;
     int success=0;
     int tries = 0;
     //char c;


     //t = 10000;
     t=1;


     for(u=0;u<(USB_BYTES_PER_VAR-2);u++) //mult to prep for # digits
     {
          //if you use more than 0 to 5 then
          //overflow and erroneous number...
          t *= 10;
          //prep for proper number of digits with zeroes preceding
     }

     if(var < 0) //if negative input variable
     {
          t /= 10; //one less digit if need to send negative sign
          u=45;//this is ASCII '-'
          do
          {
               tries++;
               success = ByteToUSB(&u);
          } while((tries<=USB_MAX_NUM_TRIES_IN) &&
          (success==APDFAIL));
```

```
            if(success==APDFAIL)goto end_of_outusbdec;

            tries=0; //rezero this if no failure

            var *= -1; //make positive

            //send the extra zero digit to make a total OF ^^^^6
            //u=0;
            //do
            //{
            //    tries++;
            //    success = ByteToUSB(&u);
            //} while((tries<=USB_MAX_NUM_TRIES_IN) &&
             (success==APDFAIL));

            //if(success==APDFAIL)goto end_of_outusbdec;

            //tries=0; //rezero this if no failure

      }

      //send out extra digit to make 6 digits
      u=0;
      do
      {
            tries++;
            success = ByteToUSB(&u);
      } while((tries<=USB_MAX_NUM_TRIES_IN) && (success==APDFAIL));

      if(success==APDFAIL)goto end_of_outusbdec;

      tries=0; //rezero this if no failure

      do { //send out the digits
            u = 0;
           while (var>=t) { var -= t; u++; }
           //c = '0' + u;
           //outdec(SCI0DR, u); //for debug to check digit
           do
            {
                  tries++;
                  success = ByteToUSB(&u);
            } while((tries<=USB_MAX_NUM_TRIES_IN) &&
            (success==APDFAIL));

            if(success==APDFAIL)goto end_of_outusbdec;

            tries=0; //rezero this if no failure

           t /= 10;
      } while (t>0);

end_of_outusbdec:
;

if(success==APDFAIL)
{
      outsci(SCI0DR, "OUDx");
```

```
     return APDFAIL;
} else
     return OK;

}

/*********************************************************/
int inusbdecfloat(float* var)
{

     //this should work for Frac16 as well
     //send out digits of an int as chars via port A
     //assumes that values are not large - need support for positive
     //and negative integers
     //also note proper number of digits
     int i, deccounter;
     int success=0;
     //char c;
     float sign = 1.0f;
     float floatval = 0.0f;
     float pow10 = 1.0f;
     int countdecplaces = 0;
     unsigned int dig = 0;
     int tries = 0;

     //t = 10000;
     pow10=1.0f;
     deccounter = 0;
     i=0;

     for(i=0;i<(USB_BYTES_PER_VAR-1);i++) //mult to prep for # digits
     {
          //if you use more than 0 to 5 then overflow
          //and erroneous number...
          pow10 *= 10.0;
          //prep for proper number of digits with zeroes preceding
     }

     for(i=0;i<USB_BYTES_PER_VAR;i++)
     {
          do
          {
               tries++;
               success = ByteFromUSB(&dig);
          } while((tries<=USB_MAX_NUM_TRIES_IN) &&
          (success==APDFAIL));

          if(success==APDFAIL)goto end_of_inusbdecfloat;

          tries=0; //rezero this if no failure

          if(dig==USB_DEC_PT)
          {
               countdecplaces=1;
          }else if(dig==USB_NEG_SIGN)
          {
               sign *= -1.0;
          }else
          {
```

```
                floatval = floatval + (dig * pow10);
                if(countdecplaces!=0)
                     deccounter++;
                pow10 = pow10 / 10.0;
           }
      }

      floatval *= sign;
      pow10=1.0;
      if(sign==-1.0)pow10=10.0; //-13 was seen as -130. before this
      for(i=0; i<=deccounter; i++)
      {
           pow10 *= 10.0;
      }
      floatval /= pow10;
      //this sets final proper position of decimal place

      *var = floatval;

end_of_inusbdecfloat:

if(success==APDFAIL)
      return APDFAIL;
else
      return OK;

} //inusbdecfloat



/*********************************************************/
int outusbdecfloat(float * floatin)
{

      //send out digits of a float as chars via port A

      //var decs
      float rounddivisor = 5.0;
      int hasnegsign = 0;
      int digitplace = 0;
      int decdigit = 0;
      int num10divs = 0;
      float multcounter = 10.0;
      int totdigout = 0;
      int prevaccum = 0;
      int value = 0;
      float copyfloat = 0.0;
      unsigned int dig;
      int i;
      int success = APDFAIL;
      int tries = 0;

      //debug trial numbers
      copyfloat = *floatin;
           //floatin *= 3.999f;
           // to preserve rr value in calling function


      //digit = static_cast<unsigned int>(100.0f * floatin);
```

```
//process

//** first first check for negative and then abs
if(*floatin < 0.0)
{
     dig = (unsigned int)USB_NEG_SIGN;
     success = ByteToUSB(&dig);
     *floatin = *floatin * -1.0;
     hasnegsign=1;
     //totdigout = USB_BYTES_PER_VAR - 1;
} else {
     //totdigout = USB_BYTES_PER_VAR;
}

//** first count number power of ten
while((int)(*floatin / multcounter) != 0)
{
     num10divs++;
     multcounter *= 10.0;
}
//** loop to output each digit - limited to -99.00
//or 999.00 for DIGOUTPRECIS=2
totdigout = USB_BYTES_PER_VAR - num10divs - 1 - hasnegsign;
for(i = 0; i < totdigout; i++)
//to prep for rounding:

rounddivisor /= 10.0;
// - create 5 digit to add to LSDig

*floatin += rounddivisor;
//add the 5 to LSDig

*floatin /= multcounter;
//start with decimal at beginning

multcounter = 10.0;
//reset multcounter
//OutFile << "floatin: " << floatin << endl;

//now reset total digits out for proper USB functioning
//tries=0;
totdigout = USB_BYTES_PER_VAR - hasnegsign;
decdigit = totdigout - num10divs - 1;
for(i = 0; i < totdigout; i++) //just changed this to < from <=
{
     //OutFile << "totdigout: " << totdigout << endl;
     digitplace = totdigout - i;
     if(digitplace == (decdigit)) //might need decimal place
     {
          dig = (unsigned int)USB_DEC_PT;
          do
          {
               tries++;
               success = ByteToUSB(&dig);
          } while((tries<=USB_MAX_NUM_TRIES_IN) &&
           (success==APDFAIL));

          if(success==APDFAIL)goto end_of_outusbdecfloat;
```

```
                tries=0; //rezero this if no failure

                goto end_of_outusbdecfloat_loop;
            }


            prevaccum *= 10;
            value = (long)(*floatin * multcounter);
            //output'digit' value here
            //for decimal place on MAX and LED chips use
            //digitplace == DPDIG

            //c = '0'+digit;
            dig = (unsigned int)(value - prevaccum);
            //OutFile << "floatin * multcounter: " << (value)
            //<< ", (short)(floatin * multcounter): " << digit << endl;
            multcounter *= 10.0;
            prevaccum = value;

            do
            {
                tries++;
                success = ByteToUSB(&dig);
            } while((tries<=USB_MAX_NUM_TRIES_IN) &&
             (success==APDFAIL));

            if(success==APDFAIL)goto end_of_outusbdecfloat;

            tries=0; //rezero this if no failure

            //success = ByteToUSB(&dig);

            /*
            if(USEMAX7221)
            {
                digitplace = totdigout - i;
                if(digitplace == 4 && digit == 0)digit = 0x0f;
                //if MSD is zero, blank it
                if(digitplace == DPDIG)digit |= 0x80;
                //turn on dec pt if necess
                WriteOutDigitMAX7221(digit, digitplace);
            }
            else
            {
                digitplace = totdigout - i - 1;
                WriteOutDigit4511(digit, digitplace);
            }
            //LoopWait(1000);
            */
        end_of_outusbdecfloat_loop:
        ;
        }

        *floatin = copyfloat; //if debug, return original ref value to
    original state

    end_of_outusbdecfloat:
    ;
```

```
if(success==APDFAIL)
     return APDFAIL;
else
     return OK;

}

/*********************************************************/
void SetupPWM(void)
{

       //Set up the PWM for motor control based
       //on macros for PWM frequency and config

            //(*(char *) 0x0fa1) = 0xf413;
            /*PWMA_CTL = 0x0001;
            PWMA_OUT = 0x8000;        //enable output pads of PWM
            PWMA_MCM = 0x2710;
            //period is 10000 clock cycles (IPBus = 40MHz

            // and desired is 4000 = 40M/10000
            PWMA_VAL0 = 0x2710;       //pulse width in clock periods
            PWMA_PMDISMAP1 = 0x0000;
            PWMA_PMDISMAP2 = 0x0000;
            PWMA_CFG = 0x100e;
            //edge-aligned, independent channels
            (*(char *) 0x0e0c) = 0x0000; //zero deadtime
            PWMA_CTL = 0x0003;
            */


            //This works...
            PWMA_CFG = 0x100e;//PWMA_CFG | 0x100e;//bfset 0x100e
            PWMA_PMDISMAP1 = 0x0000;//PWMA_PMDISMAP1 & (~0xffff);
            //bfclr 0xffff
            PWMA_PMDISMAP2 = 0x0000;//PWMA_PMDISMAP2 & (~0x00ff);
            //bfclr 0x00ff
            PWMA_PMFSA = 0x0055;//PWMA_PMFSA | 0x0055;
            PWMA_OUT = 0xbc00;//PWMA_OUT | 0xBC00;
            PWMA_MCM = 0x0400;
            PWMA_VAL0= 0x0000;
            //PWM duty cycle as number of clock cycles
            PWMA_CTL = PWMA_CTL | 0x0002;
            PWMA_CTL = PWMA_CTL | 0x0001;

            //Try setting up PWM as Timer D Chan 2 output
            //asm (bfset #$0100,sr);  //level 0 interrupt mask
            //asm (bfclr #$0200,sr);
            //level 1 int mask        /* allow lowest priority */
            /*
            ITCN_GPR7 = TMRD2_IPL;
            TMRD2_CTRL = 0x0000;
            TMRD2_CMP1 = 0x0200;
            TMRD2_CMP2 = 0x0000;
            TMRD2_LOAD = 0x0100;
            TMRD2_CNTR = 0x0100;
            asm (bfset #$0003,x:$0d37);
            //TMRD2_SCR = TMRD2_SCR | 0x0003;
```

```
            //allow it to be set on output pin, OFLAG that is
            TMRD2_CTRL=0x3e33;//0x5e14;
            //use alt cmp1 and cmp2 regs counting up and down
            //from load value
            */
            /*
            ITCN_GPR9 = TMRB0_IPL;
            // Group Priority for Timer B, channel 0
            TMRB0_CTRL=0x0000;
            TMRB0_CMP1=comparevalue;//0xffff;
            TMRB0_CMP2=0x0000;
            TMRB0_LOAD=comparevalue;//0xffff;
            TMRB0_CNTR=comparevalue;
            TMRB0_SCR=0x4000;
            TMRB0_CTRL=0x3e30;
            */
}

/*******************************************************/
void SetPWMDuty(int* duty)
{

      //Adjust the PWM duty cycle based on int* duty


      PWMA_VAL0 = *duty;
      PWMA_CTL = PWMA_CTL | 0x0003;
      //if(TALK)outsci(SCI0DR, "\15\n\SetPWMDuty:\15\n");
      //if(TALK)outdec(SCI0DR, *duty);

}

/*******************************************************/
void SetupTimerInterrupt(const int comparevalue)
{

      //Set up the timer interrupt based on macros
      // - interrupt is triggered when counter reaches
      // the const int comparevalue

      asm (bfset #$0100,sr);    //level 0 interrupt mask
      asm (bfclr #$0200,sr);    //level 1 int mask
      /* allow lowest priority */
      //above means allow all interrupts
      IPR_W = IPR_W | IPRVAL_NOIRQ;
      // set Interrupt Priority Register
      //ITCN_GPR7 = TMRD0_IPL|TMRD1_IPL;
      /*
      ITCN_GPR10 = TMRA0_IPL;
      // Group Priority for Timer D, channel 0
      TMRA0_CTRL=0x0000;
      TMRA0_CMP1=0x7fff;
      TMRA0_CMP2=0x0000;
      TMRA0_LOAD=0x7fff;
      TMRA0_CNTR=0x7fff;
      TMRA0_SCR=0x4000;
      TMRA0_CTRL=0x3e30;
      */
      /*
```

```
    // stop timer 0
    TMRD0_CTRL=0x0000;
    TMRD1_CTRL=0x0000;
    //outscihex(SCI0DR,TCFIE);

    TMRD1_CMP2 = 0x0000;  //CMP2VAL;
    TMRD1_CMP1 = 0xffff;
    TMRD1_LOAD = 625; //TMRINITVAL;
    TMRD1_CNTR = 625;

    //TMRD0_CMP1 = 500;  //CMP1VAL;
    // count up comparison value
    TMRD0_CMP2 = 0x0000;  //CMP2VAL;
    //count down comparison value
    TMRD0_LOAD = 0x0fff;  //TMRINITVAL;    // reload value
    TMRD0_SCR  = 0x4000; //TCFIE;        // enable compare interrupt
    //TMRD1_SCR = 0x4000;
    TMRD0_CTRL = 0xea30;  //SETUPTMRD0;
    // count IPBus/128, repeat, reinit, count up
    TMRD1_CTRL = 0x3e30; //SETUPTMRD1;
    */

    ITCN_GPR9 = TMRB0_IPL;
    // Group Priority for Timer B, channel 0
    TMRB0_CTRL=0x0000;
    TMRB0_CMP1=comparevalue;//0xffff;
    TMRB0_CMP2=0x0000;
    TMRB0_LOAD=comparevalue;//0xffff;
    TMRB0_CNTR=comparevalue;
    TMRB0_SCR=0x4000;
    TMRB0_CTRL=0x3e30;

} //SetupTimerInterrupt

/*********************************************************************/
void SetupTimerCheck()
{

     //This is used to set up a timer for
     //time profiling code execution - if
     //count can b examined to determine
     //how long a segment of code has taken
     //to execute.  This is used as part of
     //the testing and init mode to ensure proper
     //sample acquisition time.

    asm (bfset #$0100,sr);   //level 0 interrupt mask
    asm (bfclr #$0200,sr);   //level 1 int mask
    /* allow lowest priority */
    //above means allow all interrupts
    IPR_W = IPR_W | IPRVAL_NOIRQ;
    // set Interrupt Priority Register

    ITCN_GPR10 = TMRA0_IPL;
    // Group Priority for Timer A, channel 0
    TMRA0_CTRL=0x0000;
    //TMRA0_CMP1=0x7fff;
    //TMRA0_CMP2=0x0000;
    TMRA0_LOAD=0x0000;
```

```
      TMRA0_CNTR=0x0000;
      TMRA0_SCR=0x0000;
      //No interrupt compares or anything like this
      //TMRA0_CTRL=0x3e00; //0x3c00 = ip / 64


} //SetupTimerCheck


/********************************************************/
void GetADSSamples(short* first, short* second)
{
      //This acquires two samples from channels 0 and 1 on the
      //ADS ADC chip - either ADS7824 or 7825 depending on
      //selected bits in macros
      //Most recently confirmed for 7825 16-bit testing

      //consts

      //var decs
      short processok = 0;
      long value = 0x00000000;
      short i = 0;
      long value2 = 0x00000000;

      //process

      //outsci(SCI0DR, "GetADSSamples");

      //may want to use controlled _CS now to share port IO
      //thus use _CS to make _CS low during conversions
      //notes that it needs 10ns time between _CS low and _RC going
      //this corresponds to 10Mhz or 4 to 8 instruction cycles...

      //set direction of PA I/O
      PADDR = ADS_DDR;

      //first clear the CS bit to the ADC
      SELECTADS;

      //first channel
      InitConv(1);
      //InitConv(0); //doesn't work in this order do 1 first....

      for(i = 0; i < NADSBITS+PADREADBITS; i++)
      {
            PADR = (PADR R_C_1) DCLK_1;
            //this initiates sync pulse first, then


            //then second rising edge msb
            value <<= 1;

            if(READADSdata)
            //(short)(_inp(status)&0x20 STATMASK) -
            //data read on rising edge
                        value |= 0x01;
```

```
        //on DSP56803 this outputs a 640 ns pulse

        //max freq for ADS7825 is 10 MHz

        PADR = (PADR R_C_1) DCLK_0;

        //LoopSWait(1);
        asm(nop);
        //adjust this to get proper duty cycle (20 to 70%
        //dclk high time)
        asm(nop);
        //five nops in DSP56803 gives about 50%
        asm(nop);
        asm(nop);
        asm(nop);
}


value2 = value;
value >>= RTSHIFTBITS;
value &= ~(LEADDUMMYBITMASK);
//get rid of first junk bits (4)


*second = (short)value;
//used to be first, but reversed the data
value = 0x00000000;

//Need to wait (conservative est) 2
//usec between convert command and last DCLCK
LoopSWait(1); //this and prev commands gives ~23 usec

//Do second channel
InitConv(0);
//InitConv(1); //doesn't work in this order....

for(i = 0; i < NADSBITS+PADREADBITS; i++)
{
        PADR = (PADR R_C_1) DCLK_1;
        //12

        value <<= 1;
        if(READADSdata)
        //(short)(_inp(status)STATMASK)
                value |= 0x01;
        //high pulse is about 3.5 usec

        PADR = (PADR R_C_1) DCLK_0;
        //13 -> MSB to LSB

        //LoopSWait(1);
        asm(nop);
        asm(nop);
        asm(nop);
        asm(nop);
        asm(nop);

}
value >>= RTSHIFTBITS;
```

```
     value &= ~(LEADDUMMYBITMASK);
     *first = (short)value; //used to be second...

     //R/_C should remain high and DCLK should remain
     //low during no activity
     PADR = (PADR R_C_1) DCLK_0;

     //then de-select the chip to allow bus to be used
     //for USB if necessary
     DESELECTADS;

     //outdec(SCI0DR, *first);
     //outdec(SCI0DR, *second);

} //GetADSSamples


/*******************************************************/
void InitConv(const short NextChan)
{

      //Initiate ADC conversion on the ADS7825/4 with
      //the appropriate logic

     //short processok = 0;

     //assumes this proc is entered with RC set high and CS tied low

     //Init conversion with RC low pulse
     //outsci(SCI0DR, "IC");

     PADR = (PADR R_C_0) DCLK_0;

     PADR = (PADR R_C_1) DCLK_0;

     if (NextChan == 1)
     {    SETADSADDRCH1;            //(short)_outp(data, 0x01)
          //outsci(SCI0DR, "Ch1");
     }else{
          SETADSADDRCH0;
          //outsci(SCI0DR, "Ch0");
     }

     while(!(READADSBUSY)){}
     //this should be 25usec worst case max

} //InitConv

/*******************************************************/
void LoopSWait(const long numcycles)
{

      //Short wait loop - compare with LoopWait
      //empty loops are normally stripped out
      //by compiler

     long i = 0;
     for (i = 0; i < numcycles; i++)
     {
```

```
            //dummy loop
            // more nop's for consistence across optimization
            asm (nop);
            //asm (nop);
            //asm (nop);
            /*asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);*/
      }
}

/***********************************************************/
void LoopWait(const long millisec)
{

       //Wait loop
       //Empty loops are otherwise stripped out by
       //compiler

      //consts
      long cycles = CLOCKSPD * 100;
      long k= 0;
      long j =0;

      for(k = 0; k < millisec; k++)
      {
            for(j=0; j < cycles; j++)
            {
                  //wait loop
                  // more nop's for consistence across optimization
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            asm (nop);
            }
      }
      return;
```

```
}


/**************************************************/
void ADSampleToVoltage(short* digital1, short* digital2, float* val1,
float* val2)
{

	 //Convert ADC sample to voltage if needed
	 //not used in final prototype

	//static float prevval1 = 0.0f;
	//static float prevval2 = 0.0f;

	//*val1 = (((float)*digital1) * (float)ADCAL);
	//*val2 = (((float)*digital2) * (float)ADCAL);
	/*
	if((val1 > MAXVOLTAGE) || (val1 < MINVOLTAGE))
		val1 = prevval1;

	if((val2 > MAXVOLTAGE) || (val2 < MINVOLTAGE))
		val2 = prevval2;
	*/

	//prevval1 = *val1;
	//prevval2 = *val2;

	return;
} //ADSampleToVoltage


/*****************************************************************
*/
void ADVoltagesToScaledValues(Frac16* pressval, Frac16* flowval)
{

	 //Convert or scale ADC values
	 //in this current version, only offset
	 //is subtracted here, as scaling is applied
	 //later in order to retain integer values
	 //for as long as possible, reducing needed
	 //buffer sizes.

	*pressval = __sub(*pressval, pressoff);// * pressspan;
	*flowval = __sub(*flowval, flowoff);// * flowspan;
	//if(LOGdata)OutFiledata << dec << "scaled press, flow:
	//"<< pressval << " " << flowval;

	//YES: __sub(s1, s2) is s1-s2

} //ADVoltagesToScaledValues


void StartProfileTimer()
{
	//Set the timer for going:
	TMRA0_CTRL = 0x3e00;
	//Count rising edges, IPBus/128 rate, count up
} //StartProfileTimer
```

```
void StopProfileTimer()
{
     //Check the counter register
          TMRA0_CTRL = 0x0000; //stop the counter
          //pressAD = TMRA0_CNTR;
          outsci(SCI0DR, "Timer stopped: ");
          outscihex(SCI0DR, TMRA0_CNTR);
          TMRA0_CNTR = 0x0000; //reset the counter register to 0
} //StopProfileTimer

void TimeProcStreamMode()
{

      //This is the main timer interrupt called
      //function for simple data collection for
      //rezeroing or for streaming data to the
      //USB PC host.

     short pressAD = 0;
     short flowAD  = 0;
     //assume 16-bit returned dig val

     int time = 0;
     int success = OK;

     //float flttest = 0.0f;
     //int inttest = 0;

     unsigned int dummy = 0;

     if((mode&USBATTACHMASK)!=0) //if USB is attached
     {
          //Start timer for debug
          //outsci(SCI0DR, "USA ");
          time=0;
          TMRA0_CNTR=0;
          STARTCHECKTIME;
     }

     //Acquire samples
     GetADSSamples(&pressAD, &flowAD);

     //flttest = (float)pressAD;


     g_accumfloat1 += FracToFloat(pressAD);
     g_accumfloat2 += FracToFloat(flowAD);
     //outdec(SCI0DR, pressAD);

     //inttest = (int)g_accumfloat1;

     if((mode&USBATTACHMASK)!=0)
     //if USB is attached, send out the data
     {
          success = outusbdec((int)pressAD);
          //for debug used: (int)g_process
          success = outusbdec((int)flowAD);
          //flowAD);
```

```
            //stop timer for debug
            time = TMRA0_CNTR;
            STOPCHECKTIME;
            TMRA0_CNTR=0;
            //outscibyte(SCI0DR, 0xFE);
            //outscibyte(SCI0DR, 0x58);
            //outdec(SCI0DR, time);
            //outusbdec(time);
            ////g_process=0; // use time=0 if needed
            //tests show time to be about 420 counts for
            //this proper acquire sample
            //and transfer over USB
            if(time>=USB_HOST_TOO_SLOW_STREAM)
            {
                  if(DEBUG)outsci(SCI0DR, "USTSx");
                  if(USELCD)outscierr("USTSx");
                  //fill USB buffer with junk enough to
                  //cause it to send and then exit
                  dummy = USB_HOST_TOO_SLOW;
                  success = ByteToUSB(&dummy);
                  //send code for too slow host
                  dummy = USB_END_OF_VARIABLE_TRANSFER;
                  success = ByteToUSB(&dummy);
                  success = APDFAIL;
            }

      }

      g_process++;    //count number of times this has been called
      //was uncommented 1

      if(success==APDFAIL)
      {
            if(DEBUG){outsci(SCI0DR, "USfx ");}
            if(USELCD)outsciusbinfo("12");
            //Reset mode so no call is made again to this stuff
            //mode = (mode & ~STREAMMODE);
            //already done in loop at beginning
           mode |= STOPMODE;

      }

} //TimeProcStreamMode

/****************************************************************/
void TimeProcRunMode()
{


      //This is the main timer interrupt called
      //respiratory resistance reading routine
      //that grabs samples, looks for potential
      //perts, determines their integrity
      //calculates respiratory resistance,
      //performs display, averaging or
      //USB output depending on current mode
      //and settings.
```

```
//vars from APDTimetesting in MS VC++ prototypes
short pressAD = 0;
short flowAD  = 0;
//assume 16-bit returned dig val
short processok = 0;
int intprocessok = 0;
int start = 0;
int stop = 0;
static long sampcnt = 0;
static int process = 0;
static long inrrvalcnt = 0;
//num successful rr calc'd so far
static long exrrvalcnt = 0;
int isok = 0;
float curfreq = 0.0f;
float wtavfreq = 0.0f;
float inrr = 0.0f;
//respiratory resistance value calculated
float avwtrr = 0.0f;
static float inwtrr = 0.0f;
static float exwtrr = 0.0f;
float inwtrrdisp = 0.0f;
float exwtrrdisp = 0.0f;
float exrr = 0.0f;
float** pcur_rrvals = &exrrvals;
float** pcur_rrvalstemp = &exrrvalstemp;
float** pbegin_rrvals = &pbexrrvals;
float** pbegin_rrvalstemp = &pbexrrvalstemp;
float* prr = &exrr;
float* pwtrr = &exwtrr;
long* prrvalcnt = &exrrvalcnt;
//int* pcallmode = (int*)dwUser;
//int  copymode = *pcallmode;
static int callmode = 0;
//static short controlport = (0x00 CTRLDIROUT MOTOR_1);
//float fltpress = 0.0f;
//float fltflow = 0.0f;
float floatval = 0.0f;
int success = OK;
int lcdplacecounter = 0;
static int last10 = 10;
int time = 0;

//if(TALK)outsci(SCI0DR, "\15\nCalled TimeProcRunMode");

//monitor loop execution time
//This is commented out because it is only for debugging,
//testing etc.
//see end of this program
/*
time=0;
TMRA0_CNTR=0;
STARTCHECKTIME;
*/

//Rezero all statics etc. if this is the first time
//this was called
if(g_process==0)
```

```
{
     exwtrr=0.0;
     inwtrr=0.0;
     sampcnt=0;
     process=0;
     inrrvalcnt=0;
     exrrvalcnt=0;
}

if(g_process==0)g_process=1;

//-----------------------------------------------------------------
//** GET BOTH press and flow samples.
//** Press assumed chan 0, flow chan 1
GetADSSamples(&pressAD, &flowAD);

//-----------------------------------------------------------------
//** CONVERT samples to voltages
//ADSampleToVoltage(&pressAD, &flowAD, &pressval, &flowval);
pressval = pressAD;
flowval = flowAD;

//-----------------------------------------------------------------
//** CONVERT Voltages to Scaled values
ADVoltagesToScaledValues(&pressval, &flowval);
//Now this converts Frac16 just by offset

//-----------------------------------------------------------------
//** FILTER new sample in IIR filter
DualFilterLoop4Pole100_500(&pressval, &flowval, FALSE);

//-----------------------------------------------------------------
//** PLACE scaled values into proper circular buffer position
*press++ = pressval;
*flow++ =  flowval;

//floatval = (float)pressval;
//outdecfloat(SCI0DR, floatval);
//floatval = (float)flowval;
//outdecfloat(SCI0DR, floatval);

//-----------------------------------------------------------------
//** CALCULATE RESISTANCE at this press - flow pair
//Now use scaled values for rd to facilitate minrd and
//maxrd selection for finding
//most common rd value later...this then uses values
//already shown to work in
//earlier LPT port prototype of the system
g_accumfloat1 = (float)pressval;
g_accumfloat2 = (float)flowval;

*rd = g_accumfloat1/g_accumfloat2; //divide
*rd = *rd * pressspan / flowspan; //scale

//have potential for divide by zero here with the frac. numbers:
//if(*rd > MAXRDALLOW || *rd < MINRDALLOW)*rd=0.0;
if(g_accumfloat2==0.0)*rd=0.0; //this prevents hang on
//"inf" in outusbdecfloat
//floatval = *rd++;
```

```
rd++;

//StopProfileTimer();
/*if(DEBUG)
{
outsci(SCI0DR, "\15\nFirst sample: ");
outdecfloat(SCI0DR, fltpress);
outdec(SCI0DR, pressAD);
outsci(SCI0DR, "\15\nSecond sample: ");
outdecfloat(SCI0DR, fltflow);
outdec(SCI0DR, flowAD);
}*/
//StopProfileTimer();

//-----------------------------------------------------------------
//** CHECK CIRC BUFFER - if past end, circulate to beginning
if(press > (pbpress + SAMPBUFLEN - 1))
{
     press = pbpress;
     flow = pbflow;
     rd = pbrd;
     //drd = pbdrd;
     //Don't increment drd = not a TRUE buffer
     //- only rewritten array
}

//-----------------------------------------------------------------
//** CHECK for full buffer (calls >= PERFBUFFERLEN) -
//** matters only at startup
//** if buffers are full, then set 'process' to begin
//** processing buffers
if(DEBUG)  //increment always if DEBUG to keep track of
     //time for simulated breath
     sampcnt++;
if(!process)
{
     if(!DEBUG)
          sampcnt++;
     if (sampcnt >= SAMPBUFLEN)
          process = 1;
}

//----------------------------------------------------------------
//** if BUFFERS FULL with samples, then PROCESS BUFFER
if(process >= 1)
{
     //cout<< "process in run entered." << endl;

     //-----------------------------------------------------------
     //** COPY CIRC buffers into arrays at proper endpoints
     //OutFile << "About to copy arrays..." << endl;
     CopyCircArray(rd, rdtemp, pbrd, pbrdtemp, SAMPBUFLEN);
     CopyCircArrayFrac(press, presstemp, pbpress, pbpresstemp,
          SAMPBUFLEN);
     CopyCircArrayFrac(flow, flowtemp, pbflow, pbflowtemp,
          SAMPBUFLEN);

     //-----------------------------------------------------------
     //** COMPARE - DEBUG only - Use circ buffer methods for
```

```
//** exec time check
//DifferentiateArrayCirc();

//------------------------------------------------------------
//** DIFFERENTIATE rdtemp array
DifferentiateArray(rdtemp, drd, SAMPBUFLEN);

//------------------------------------------------------------
//** FIND MCV RD (Most common value of device resistance)
intprocessok = FindMostCommonValue(rdtemp, SAMPBUFLEN,
     MINRD, MAXRD, NUMBINS, &rthres);
if(intprocessok == APDFAIL)
     rthres = NOMINALRTHRES;
//OutFile << "rthres: " << rthres << endl;

//------------------------------------------------------------
//** DETERMINE if FULL PERT present in data and greater than
//** MINPERTLEN
intprocessok = FullPertPresent(rdtemp, drd, SAMPBUFLEN,
     &start, &stop, &rthres);

if(intprocessok != APDFAIL)
{
     //if(TALK)outsci(SCI0DR, "PertFnd");
     //if(TALK)outdec(SCI0DR, stop-start);
     //------------------------------------------------------
     //** DETERMINE if FLOW is CONSISTENT DIRECTION and
     //** ABOVE THRESHOLD
     //** If consistent, then calc respiratory resistance
     intprocessok = ConsistentDirection(flowtemp,
          SAMPBUFLEN, &start, &stop);
     //outsci(SCI0DR, "Consistent Direction? ");
     //outdec(SCI0DR, intprocessok);
     if(DEBUG)outsci(SCI0DR, "\15\n");
     if(intprocessok == INH)
     //at each call of this procedure, these pointers
     //are initialized to ex vars
     {
          pcur_rrvals = &inrrvals;
          pbegin_rrvals = &pbinrrvals;
          pcur_rrvalstemp = &inrrvalstemp;
          pbegin_rrvalstemp = &pbinrrvalstemp;
          prr = &inrr;
          prrvalcnt = &inrrvalcnt;
          pwtrr = &inwtrr;
          if(DEBUG)outsci(SCI0DR, "I ");
     }
     if(intprocessok != APDFAIL)
     {
          //--------------------------------------------------
          //** IF OK, CALC RESPIRATORY RESISTANCE
          //** Calculate respiratory resistance
          intprocessok = CalcRespResist(presstemp,
               flowtemp, SAMPBUFLEN, &start, &stop,
               &intprocessok, prr);

          if(intprocessok != APDFAIL)
          {
               //Toggle Yellow LED (or some LED
```

```
//to show ack
if((PEDR & 0x0008) == 0)
{
     PEDR = PEDR | 0x0008;
}else{
     PEDR = PEDR & ~(0x0008);
}

//Check pert frequency
//SAMPFREQ: because run mode
//sample rate is 250 Hz
curfreq = (SAMPFREQ / 2) / (stop - start);
//curfreq >>= 1; //divide by two
*freq++ = curfreq;

*(*pcur_rrvals)++ = *prr;
//if(DEBUG)cout<< "rr pt: " << *prr;

floatval = *prr;
//set value to USB host as the rr val.
*pwtrr += *prr;
//accum rrs in proper number -
//this will be replaced
//by a weighted average value in the
//below code, if that
//option is selected...otherwise, these
//values are accum
//until max num perts in both directions is
//collected
//then values are divided, avged, and
//displayed.
//assumes that in CalcRespResist, pressval
//and flowval are set to
//actual measured flow and pressure values
//at the perturbation value


if((*prrvalcnt) < RRVALLEN)
{
     (*prrvalcnt)++;
     //floatval = *prr; //*pwtrr;
     //for stream to host

     if((*prrvalcnt) >= RRVALTILLDISP)
     {
     //*pwtrr = *prr;
     if(USEWEIGHTEDAVERAGE)
     {
     *pwtrr = WtAverage((*pbegin_rrvals),
            (*prrvalcnt), LINEARAVG);
            lcdplacecounter =
            OutputRespResistValue(pwtrr);
            }else{
            lcdplacecounter =
            OutputRespResistValue(prr);
            }

            wtavfreq = WtAverage(pbfreq,
            (*prrvalcnt), LINEARAVG);
```

```
                //LED notice that not full
                //if(DEBUG)cout << ", rr ind buff
                //not full # rr vals="<<
                //(*prrvalcnt);
        }
}
else
{
        (*prrvalcnt)++;
        //added to keep incrementing number of
        //perts collected
        //floatval = *prr; //*pwtrr;
        //for stream to host
        CopyCircArray((*pcur_rrvals),
        (*pcur_rrvalstemp), (*pbegin_rrvals),
        (*pbegin_rrvalstemp), RRVALLEN);

        CopyCircArray(freq, freqtemp, pbfreq,
        pbfreqtemp, RRVALLEN);

        //*pwtrr = *prr;
        if(USEWEIGHTEDAVERAGE)
        {
        *pwtrr = WtAverage
        ((*pbegin_rrvalstemp), RRVALLEN,
        LINEARAVG);
        lcdplacecounter = OutputRespResistValue
        (pwtrr);
        }else{
        lcdplacecounter = OutputRespResistValue
        (prr);
        }
        wtavfreq = WtAverage(pbfreqtemp,
        RRVALLEN, LINEARAVG);

        //LED notice that ok
}

//Adjust PWM frequency if necessary
//using wtavfreq ends up with too
//much delay in the control loop
//with the PWM_PD_STEP gain, get a
//lot of overshoot/ undershoot
if(SERVOPWM)
{
        if(curfreq > MAXPERTFREQ)
        {
        duty -= PWM_PD_STEP;
        if(duty < PWM_PD_MIN)duty =
        PWM_PD_MIN; //set floor for PWM
                SetPWMDuty(&duty);
        }
        if(curfreq < MINPERTFREQ)
        {
                duty += PWM_PD_STEP;
                SetPWMDuty(&duty);
        }
}//SERVOPWM
```

```
                    if(LOGdata)
                    {
                         if(pwtrr = &inwtrr)
                         {
                              inwtrrdisp = (*pwtrr);
                              exwtrrdisp = 0.0f;
                         }
                         if(pwtrr = & exwtrr)
                         {
                              inwtrrdisp = 0.0f;
                              exwtrrdisp = (*pwtrr);
                         }
                    }

                    if((*pcur_rrvals) > ((*pbegin_rrvals) +
                    RRVALLEN - 1))
                    {
                         (*pcur_rrvals) = (*pbegin_rrvals);
                         freq = pbfreq;
                         //if(VERBOSE) cout << "Reset rrvals
                         pointer to beginning: " <<
                         (*pcur_rrvals) << endl;
                    }
                    //if(DEBUG)DebugPrintArray((*pbegin_rrvals),
                    // (*prrvalcnt), "RR vals");
                    //if(DEBUG)cout << ", wt avg wtrr: " <<
                    //(*pwtrr) << endl;
               }
          } //if(intprocessok != APDFAIL)

          //-----------------------------------------------------
          //** ZERO OLD PERT data POINTS to go to the next
          //OutFile << "Outer start: " << start << ", Outer
          //stop: " << stop << endl;
          intprocessok = ZeroOldPertRDPoints(rd, pbrd, (pbrd +
          SAMPBUFLEN - 1), &start, &stop);
     }

     process++;
     if(process==0x00ff)process=0x00fe;
     //without this, it wraps to -32xxx
     //and thus inhibits further processing
     //g_process = process;
}

if((mode&USBATTACHMASK)!=0)
//if USB is attached, send out the data
{
     //assumes that in CalcRespResist, pressval and flowval are
     //set to
     //actual measured flow and pressure values at the
     //perturbation value
     success = outusbdec((int)pressval);
     //use instead of pressAD to get offset corrected value
     success = outusbdec((int)flowval);
     success = outusbdecfloat(&floatval);
}

if((mode&USBATTACHMASK)==0)
```

```
//if USB not attached, look for max num perts and average
{                                   //when max is reached
     if(MAXNUMPERTS > 0)
     {
          /*if((inrrvalcnt >= MAXNUMPERTS) && (exrrvalcnt <
     MAXNUMPERTS) && (g_process<2))
          {
               displayvalue1 = inwtrr / (float)MAXNUMPERTS;
               g_process=2;
          }

          if((inrrvalcnt < MAXNUMPERTS) && (exrrvalcnt >=
          MAXNUMPERTS))
          {

          }*/


          if((inrrvalcnt >= MAXNUMPERTS) && (exrrvalcnt >=
          MAXNUMPERTS))
          {
               //show avg RR vals and then stop the run mode
               //displayvalue1 = inwtrr / (float)inrrvalcnt;
               //displayvalue2 = exwtrr / (float)exrrvalcnt;
               inwtrr /= (float)inrrvalcnt;
               exwtrr /= (float)exrrvalcnt;

               avwtrr = (inwtrr + exwtrr) / 2.0;

               if(DEBUG){
                    outsci(SCI0DR, "MNP ");
                    OutputRespResistValue(&avwtrr);
               }

               if(USELCD){
                    LCDClearScreen();
                    outsci(SCI0DR, "Ave RR:");
                    LCDGoToLineTwo();
                    outdecfloat(SCI0DR, avwtrr);
                    CLS1;
                    outscibyte(SCI0DR, 0x54);


                    //turn on LCD backlight (for three minutes
                    //max)
                    CLS1; //prep for command;
                    outscibyte(SCI0DR, 0x42);
                    // 46 = off; 42 = backlight on
                    outscibyte(SCI0DR, 0x03);
                    //zero means inf minutes
               }

               mode |= STOPMODE;

          }else{ //show percent completed on screen
               if((inrrvalcnt>=RRVALTILLDISP) && (exrrvalcnt >=
               RRVALTILLDISP))
               {
```

```
                    intprocessok = inrrvalcnt < exrrvalcnt ?
                    inrrvalcnt: exrrvalcnt;
                    //(inrrvalcnt + exrrvalcnt)/2;
                    //avg num of perts so far
                    intprocessok *= 100;
                    //percent of total needed

                    intprocessok /= MAXNUMPERTS;
                    //fraction of total needed

                    if(intprocessok >= last10){
                         CLS1; //prep for command
                         outscibyte(SCI0DR, 0x47);
                         //go to cursor position
                         outscibyte(SCI0DR, 0x05);
                         //column 5 of 8
                         outscibyte(SCI0DR, 0x01);
                         //row 1

                         outsci(SCI0DR, ":");
                         //no digit there
                         //for very large number of one
                         //type of pert...
                         //if(last10 > 90){
                         //    last10=90;}
                         outscibyte(SCI0DR, ((last10/10)
                         +0x30));
                         //first digit - add 0x30 to make
                         //ascii 1 = digit number 1
                         outsci(SCI0DR, "0%");

                         last10 += 10;
                         //increment to next 10 percent mark

                         CLS1;
                         outscibyte(SCI0DR, 0x47);
                         outscibyte(SCI0DR, lcdplacecounter+1);
                         outscibyte(SCI0DR, 0x02);
                         //go back to second line at
                         //last cursor position
                    }
               }
          }
     } //MAXNUMPERTS>0
} //mode&USBATTACHMASK==0

if(success==APDFAIL)
{
     if(DEBUG)outsci(SCI0DR, "USfx ");
     //if(USELCD)outscierr("USFx ");
     if(USELCD)outsciusbinfo("13");
     //Reset mode so no call is made again to this stuff
     //mode = (mode & ~STREAMMODE);
     //already done in loop at beginning
    mode |= STOPMODE;

}

////if(LOGdata)OutFiledata<< " " << (*prr);
```

```
	//log individual rr value
	////if(LOGdata)OutFiledata<<" "<<(*pwtrr);
	//log weighted avg rr value

	//if(LOGdata)OutFiledata<< " " << inrr << " " << exrr << " "
	//                        << inwtrrdisp << " " << exwtrrdisp <<
	//" "
	//                        << (*pwtrr) << " " << curfreq << " "
	//<< wtavfreq;
	//if(LOGdata)OutFiledata<<endl; //next line of data file for
	//whatever entries placed within

	/*
	//if a maximum number of rr measurements has been selected i.e.
	for consistent measurement,
	//chech to see the criterion has been met and if so, stop the
	measurement
	if(MAXNUMPERTS > 0)
	{
		if((inrrvalcnt > MAXNUMPERTS) && (exrrvalcnt >
		MAXNUMPERTS))
		{
			displayvalue1 = inwtrr;
			displayvalue2 = exwtrr;
			(*pcallmode) &= ~(RUNBIT);
					//turn off run bit
			(*pcallmode) &= ~(CALLMODEBITS);
				//rezero callmodebits
			(*pcallmode) |= ENDRUNMODE;
					//set end of run mode bit
		}
	}

	CheckButtonPress(dwUser);
	*/

//StopProfileTimer();
/*	if(DEBUG)
	{
	outsci(SCI0DR, "\15\nFirst sample: ");
	outdecfloat(SCI0DR, g_accumfloat1);
	outdec(SCI0DR, pressAD);
	outsci(SCI0DR, "\15\nSecond sample: ");
	outdecfloat(SCI0DR, g_accumfloat2);
	outdec(SCI0DR, flowAD);
	}
*/

//	if(CHECKTIME)
//	{
//		//StopProfileTimer();
//	}

	//The following is commented out because it was used to check the
	//full loop run time
	//but sends '*' to the display with each LCD update and is thus
	//clutters the display
	//we already know that at each LCD update, samples are lost.
	/*STOPCHECKTIME;
```

```
     time = TMRA0_CNTR;
     TMRA0_CNTR=0;

     if(time>0x9c4){outsci(SCI0DR, "*");}
     //0x4e2 is 2* 0x0271 thus twice counts

     //profiling timer is set for IP/64 not IP/128
     //now we are using 250 Hz in run mode so
     //check timer at 2* 0x4e2 = 0x9c4
     */

     /* clear compare status flags */
     asm (bfclr #$8000,X:TMRB0_SCR_R);

} //TimeProcRunMode

/*****************************************************/
asm Frac16 FloatToFrac(float floatin)
{

     //Convert a floating point value to a frac/int
     //adapted from motorola library

//ARTF32_TO_FXS16U:
//INPUT = a, the float
//OUTPUT = y0, the fixed value

     //;save sign
     move a1,x0
     andc #$8000,x0
     tstw x0
     beq  ssave_positive
     //;move   #$8000,x:m5               ; US 06.23.99
     move #$0001,x:$35//m5               ; US 06.23.99
     bra       ssave_done

ssave_positive:
     move #0,x:$35//m5

ssave_done:
     //; Save input
     move a0,x0
     move x0,x:$36//xlo
     move a1,x:$37//xhi

     //;check if input is zero
     andc #$7fff,a1
     tstw a1
     bne  not_zero
     tstw x0
     bne  not_zero
     movei #0,y0
     rts

     //; Check if exponent is 255
not_zero:
     move x:$37,a1//xhi,a1
     move a1,y0
     andc #$78f0,y0 //#exp_mask,y0
```

```
      cmp   #$78f0,y0 //#exp_mask,y0
      //; If it 255 (result 0), branch
      beq   exp_is_255


      movei #$7,y0
      nop
      lsrr  a1,y0,a
      nop
      nop
      andc  #$00ff,a1
      nop
      movei #127,x0
      sub        x0,a
      tstw  a1
      //; If exponent is >=0, overflow occured
      bge        max_value
      abs   a
      move  a1,x0
      move  x:$37,a1//xhi,a1
      move  #$8,y0
      cmp        #15,y0
      lsll  a1,y0,y0
      bfset #$8000,y0
      cmp   #15,x0
      ble resume411
      movei #0,y1
      bra resume412
resume411:
      lsrr  y0,x0,y1
resume412:
      ;bfset     #$4000,y1
      move  x:$36,a1//xlo,a1
      move  #$8,y0
      lsrr  a1,y0,y0
      cmp   #15,x0
      ble   resume421
      movei #0,y0
      bra   resume422
resume421:
      lsrr  y0,x0,y0
resume422:
      or         y1,y0
      //; Fall through to transfer_sign


transfer_sign:
      tstw  x:$35//m5
      beq        transfer_sign_done
      clr        a
      move  y0,a1
      neg   a
      move  a1,y0
      //;move    x:$35,y1//m5,y1
      //;or y1,y0
transfer_sign_done:
      rts

exp_is_255:
```

```
     move a1,y0
     andc #$7f,y0//frac_hi_mask,y0
     cmp       #$0,y0
     beq       hi_frac_is_zero
     bne       is_NaN


hi_frac_is_zero:
     move x:$36,y0//xlo,y0
     cmp       #$0,y0
     bne       is_NaN
     beq       max_value


max_value:
     movei #$7fff,y0
     jmp       transfer_sign


is_NaN:
     movei #$7fff,y0
     //SetInvalid
     //bfset   #$0040,x:$0000//INVALID,x:FPE_state
     rts

     //ENDSEC

} //FloatToFrac

/*************************************************************/
asm float FracToFloat(int fraction)
{

     //Convert a fraction value to a floating point value
     //Based on motorola library

//;save input
     move y0,x:$37//xhi

     ;save sign
     move y0,x0
     andc #$8000,x0
     tstw x0
     beq ssave_positive
     move #$8000,x:$32//m2

//   ;negate the input to make it positive.
     clr  b



     move y0,b1
     neg  b
     move b1,y0

     bra       ssave_done

ssave_positive:
     move #0,x:$32//m2
```

```
ssave_done:
     //;check if it is zero
     move y0,x0
     andc #$7fff,x0
     tstw x0
     bne       US0

     //;Handle 0x8000
     clr       a
     bfset#$8000,x:$32//m2
     bcc  US_ret_0
     move #$BF80,a1

US_ret_0:
     //; A was cleared before the branch to this.
     bra       end_this

US0:
     movei#$8000,x0//startmask_lsl,x0
     move y0,x:$36//m6
     move #0,a0
     movei#0,y1

US1:
     lsr       x0
     dec       y1
     move x:$36,y0//m6,y0
     and       x0,y0
     tstw y0
     beq       US1
//; Otherwise first 1 has being encountered
     bra first_one
     jmp end_this


first_one:
     move y1,y0
     add       #127,y1        //;create exponent of the float
     move y1,x:$33//m3

create_mantissa:
     //; Take absolute value of y0
     //; make it positive.
     not       y0
     add       #$1,y0


US2:
     move x:$36,y1//m6,y1
     lsll y1,y0,y1
     lsl       y1
//; Now y0 contains the value of the mantissa.
//; We proceed to split it up in accordance with IEEE format
     move y1,a1
     move #$9,y0
     lsrr a1,y0,y0
     move y0,a1
     move #$7,y0
```

```
     lsll y1,y0,y1
     move y1,a0

//; Now move the exponent to the correct place
     move x:$33,y1//m3,y1
     move #$7,y0
     lsll y1,y0,y1
     or        y1,a

//;Transfer Sign
     move x:$32,y0//m2,y0
     or        y0,a
     rts



end_this:
     rts

}

/*****************************************************/
int OutputRespResistValue(float * floatin)
{

      //Send out respiratory resistance value

     //This routine gets individual digits in base ten for
     //output routines
     //Assumes:
     // - type cast from float to long truncates float at its decimal
     //point

     //consts

     //var decs
     /*float rounddivisor = 0.5f;
     short digit;
     int digitplace = 0;
     int num10divs = 0;
     int multcounter = 10.0f;
     int totdigout = 0;
     long prevaccum = 0;
     long value = 0;*/
     float copyfloat = 0.0f;
     static int placecounter = 0;
     static short useit = 0x01;
     static int rrcounter = 0;
     static int useitback = 0x01;

     //debug trial numbers
     copyfloat = *floatin;
          //floatin *= 3.999f;
          // to preserve rr value in calling function


     //digit = static_cast<unsigned int>(100.0f * floatin);

     //process
```

```
      //** first count number power of ten

if((mode&USBATTACHMASK)==0) //if USB is not attached, send out the data
{
      if(DEBUG){
            outsci(SCI0DR, "RR: ");
            outdecfloat(SCI0DR, *floatin);
      }

      if((USELCD) && (SHOWEACHRRVAL)){
            LCDClearScreen();
            outsci(SCI0DR, "RR =");
            LCDGoToLineTwo();
            outdecfloat(SCI0DR, *floatin);
      }

      if((USELCD) && (!SHOWEACHRRVAL) && (SHOWRRASCIIINDICATOR)){

            rrcounter++;
            if(rrcounter==SHOWRRASCIIINDICATOR){
                  rrcounter=0; //reset it
                  placecounter++;
                  if(placecounter>8){
                        //should be set for not auto-wrap to stick to one
                        //line
                        useit^=0x01; //toggle useit
                        placecounter=0;
                        LCDGoToLineTwo(); //go back to start of line
                  //since auto-wrap is off
                  }
                  //if(!SHOWRRBACKLIGHTINDICATOR){
                  if(useit==0x01)outsci(SCI0DR, "-");
                  if(useit==0x00)outsci(SCI0DR, "_");
                  //}
                  useitback^=0x01; //each time display is updated,
                  //toggle backlight
                  if(SHOWRRBACKLIGHTINDICATOR){
                        if(useitback==0x01){
                              CLS1; //prep for command;
                              outscibyte(SCI0DR, 0x42);
                              // 46 = off; 42 = backlight on
                              outscibyte(SCI0DR, 0x05);
                              //zero means inf minutes
                        }
                        if(useitback==0x00){
                              CLS1; //prep for command
                              outscibyte(SCI0DR, 0x46); //off
                        }
                  }
            }
      }
}//if((mode&USBATTACHMASK)!=0) //if USB is not attached, send out the
data
//if USB is attached, don't send out data since this eats up bandwidth


      *floatin = copyfloat; //if debug, return original ref value to
original state
```

```
     return placecounter;
} //OutputRespResistValue



/****************************************************/
int ZeroOldPertRDPoints(float* curloc, float* begin, float* end, int *
start, int * stop)
{

      //Get rid of data points that have already been
      //used for a calculation

     //Assumes:
     // - start is first point above threshold
     // - stop is first point below threshold
     // - curloc is current location of pointer in array to be zeroed
     //- it
     //   refers to the location with the oldest data point (just
     //incremented
     //   after the newest data point collected).
     // - curloc is set to beginning of circ buf before entering this
     //function if it needed to wrap

     int i = 0;

     //process
     float* first = curloc + *start;
     int cnt = *stop - *start + 1;
     //for 0 to < cnt

     //need to do endpoint checking and wrapping
     if(first > end)
          first = (first - end) - 1 + begin;

     for(i = 0; i < cnt; i++)
     {
          *first++ = 0.0f;
          if(first > end)
               first = (first - end) - 1 + begin;
     }

     //OutFile << "Pert Erased: Num pts erased + 1 =" << cnt << endl;
     //DebugPrintArray(rd, SAMPBUFLEN, "rd just after erase pts.");

     return OK;
} //ZeroOldPertRDPoints



/****************************************************/
float WtAverage(float* inarray, int toteles2avg, const int type)
{

     //Calculate a weighted average of a buffer

     //toteles2avg = total elements to average starting from 0
     // - thus toteles2avg = 3 averages elements 0, 1, 2.
```

```
	//consts

	//var decs
	float eles = 0.0f;
	float numer = 0.0f;
	float denom = 0.0f;
	float fi = 0.0f;
	float* curloc = inarray + toteles2avg - 1;
	int i = 0;

	//process

	if(type == EQUALAVG)
	{
		for(i = 0; i<toteles2avg; i++)
		{
			numer += *inarray++;
			denom += 1.0;
		}
	}

	if(type == SUM)
	{
		for(i = 0; i<toteles2avg; i++)
		{
			numer += *inarray++;
		}
		denom = 1.0;
	}

	if((type == LINEARAVG) || (type == NONLINAVG))
	{
		if(type == LINEARAVG)
		{
			eles = (float)toteles2avg;
		}

		if(type == NONLINAVG)
		{
			eles = (float)toteles2avg * (float)toteles2avg;
		}

		for(i = (toteles2avg-1); i >= 0; i--)
		{
			fi = (((float)i) + 1.0f) / eles;
			numer += ((float)(*curloc--)) * fi;
			denom += fi;
		}
	}

	if(DEBUG)
		//OutFile << "Avg RR Val: " << (numer/denom) << "for num
		//pts: " << toteles2avg << endl;

	return (numer / denom);
} //WtAverage
```

```
/*****************************************************/
int CalcRespResist(Frac16* pressdata, Frac16* flowdata, const int
arraysize, int * start, int * stop, int * breathdir, float* rr)
{

     //Calculate respiratory resistance as delta press over delta
     //flow

     //assumes:
     // - start is first point above rd threshold
     // - stop is first point below rd threshold

     //consts

     //var decs
     //float x1 = 0.0f;
     //float x2 = 0.0f;
     float mf = 0.0f;
     //float bf = 0.0f;
     float mp = 0.0f;
     //float bp = 0.0f;
     float virtflow = 0.0f;
     float virtpress = 0.0f;
     float fbreathdir = 0.0f;
     int dsample = 0;
     int novolaccelindex = 0;
     int startind = 0;
     int i = 0;


     //process

     startind = (*start > 0 ? *start-1 : *start);
     dsample = *stop - startind;
                    //actual number of intervals
     //x1 = startind * DT;
     //x2 = stop * DT;

     fbreathdir = (float)(*breathdir);

     //** find flow and press fit line params
     //mf = FracToFloat(flowdata[*stop] - flowdata[startind]);
     mf = (float)(flowdata[*stop] - flowdata[startind]);
     mf = (mf) / (dsample);         //(y2-y1) / (x2-x1)
     //mb = flowdata[startind] - mf * x1;
          //y1 - mf * x1
     //OutFile << "flow slope: " << mf << endl;

     //mp = FracToFloat(pressdata[*stop] - pressdata[startind]);
     mp = (float)(pressdata[*stop] - pressdata[startind]);
     mp = (mp) / (dsample);
     //bp = pressdata[startind] - mp * x1;
     //OutFile << "press slope: " << mp << endl;

     //** find max or min of flow - point of no volume accel i.e.
     //Vdotdot = 0
     novolaccelindex = startind;
     for(i = startind; i< *stop; i++)
     {
```

```
	if(flowdata[i] * fbreathdir < flowdata[novolaccelindex] *
	fbreathdir)
		novolaccelindex = i;
}

//** calculate delta p to delta vdot ratio
virtflow  = mf*novolaccelindex + (flowdata[startind] - mf *
startind);
virtpress = mp*novolaccelindex + (pressdata[startind] - mp *
	startind);

*rr = (pressdata[novolaccelindex] - virtpress) / (virtflow -
	flowdata[novolaccelindex]);

//write value for proper sync'd output of rr with corresponding
//pressval and flowval
flowval = flowdata[novolaccelindex];

if(SENDAVRVFRATRR){//if sending virtual and actual flows at RR to
//host in USB
	pressval = (int)virtflow;//should be unscaled value still at
	//this point, as with other output values (to USB)
}else{
	pressval = pressdata[novolaccelindex];
}

//For the DSP56803 version, a final scaling needs to be applied
//to the RR
//since only offsets are adjusted prior to converting the
//fraction to float
//Use:
//              (pressspan in real per volts or dig)
// rr =   rr * ------------------------------------
//              (flowspan in real per volts or dig)
*rr = *rr * pressspan / flowspan;

//OutFile << "novolaccelindex: " << novolaccelindex << endl;
//OutFile << "virtflow: " << virtflow << endl;
//OutFile << "virtpress: " << virtpress << endl;
//OutFile << "rr within func: " << rr << endl;

//** check proper rr proportions - rr should always be positive
if((virtflow - flowdata[novolaccelindex]) == 0)return APDFAIL;
if(flowdata[novolaccelindex]<0){ //if negative number
	//and virtflow >= actual flow (less negative)
	if(virtflow >= (flowdata[novolaccelindex]-MINPERT))return
		APDFAIL;
	if(virtpress <= (pressdata[novolaccelindex]+MINPERT))return
		APDFAIL;
}
if(flowdata[novolaccelindex]>0){ //if positive number
	//and virtflow <= actual flow (less positive)
	if(virtflow <= (flowdata[novolaccelindex]+MINPERT))return
		APDFAIL;
	if(virtpress >= (pressdata[novolaccelindex]-MINPERT))return
		APDFAIL;
}
if(*rr > 0 && *rr < MAXRR)
	return OK;
```

```
     else
          return APDFAIL;

} //CalcRespResist



/*******************************************************/
int ConsistentDirection(Frac16* inarray, const int arraysize, int *
start, int * stop)
{

      //Determine if the points in a potential perturbation
      //all have same flow direction

     //looks from point just after pert starts above threshold
     //to point just before it falls below threshold to determine
     //if all points are with same flow direction and in sufficient
     //flow range

     //assumes:
     // - start is first point above threshold
     // - stop is point just below threshold
     // - exhalation is positive

     //inputs:
     // inarray should be flow array

     //consts

     //var decs
     int i = 0;
     int flowdir = INH;
     int fail = OK;
     int nextflowdir = INH;
     //int exhisneg = 0;

     //exhisneg = EXH < 0 ? -1 : 1;

     //process
     //note: multiplying array element by flowdirection forces
     //all values compared to lowflow to be positive
     flowdir = inarray[*start] EXHISNEG ? EXH : INH;
     if (inarray[*start] * (float)(flowdir) < MINFLOW)
               fail = APDFAIL;
     for(i=*start+1; i<*stop; i++)
     {
          nextflowdir = inarray[i] EXHISNEG ? EXH : INH;
          if (nextflowdir != flowdir)
               fail = APDFAIL;
          if (inarray[i] * (float)(nextflowdir) < MINFLOW)
               fail = APDFAIL;
          flowdir = nextflowdir;
     }

     //if consistent flow direction all above threshold, return the
     flow direction
     if(fail == OK)
          return flowdir;
```

```
	else
		return APDFAIL;

} //ConsistentDirection


/****************************************************************/
int FullPertPresent(float* inarray, float* dinarray, const int
arraysize, int * start, int * stop, float * thres)
{

	//Determine if a full perturbation is present
	//based on rise above and fall back below
	//threshold with proper slope at each end.

	//consts

	//var decs
	float* inarray_1 = inarray;		//to make sure that pert
	//start just after higher than thres
	int foundstart = FALSE;		//start of pert found
	int foundstop = FALSE;			//end of pert
	//bool okpertlen = 0;			//length of pert ok?
	int i = 1;

	//process
	inarray++;			//go to second array element: 1
	//(base 0) for drd have valid value
	dinarray++;				//go to second array
	//element: 1 (base 0)

	//DebugPrintArray(rdtemp, SAMPBUFLEN, "rdtemp before start
	//loc.");
	//DebugPrintArray(drd, SAMPBUFLEN, "drd before start loc.");
	while((i<arraysize-2) && (foundstart == FALSE))
	{
		//OutFile << " Start find index: " << i;
		if((*inarray > *thres) && (*inarray_1 <= *thres) &&
		(*dinarray > 0.0f))
		{
			foundstart = TRUE;
			*start = i;
			*stop = *start;
			//OutFile << "Inner start: " << start << endl;
		}
		inarray_1++; inarray++; dinarray++; i++;
	}

	if(foundstart == TRUE)
	{
		//OutFile << "Stop at end second to last of array? " << ((i
		/< arraysize-1) && (foundstop == FALSE)) << endl;
		while((i < arraysize-1) && (foundstop == FALSE))
		{
			//OutFile << " Stop find index: " << i;
			//OutFile << "drd: " << *dinarray << ", i: " << i << "
			//";
			if(*inarray <= *thres && *dinarray <= 0.0f)
			{
```

```
                    foundstop = TRUE;
                    *stop = i;
                    //OutFile << "Inner stop: " << stop << endl;
               }
               inarray++; dinarray++; i++;
          }
     }


     //Check to see that start and stop were found for pert.  If  //
length is less than min
     //pert length, then zero the rd points to get rid of the useless
     //data.

     if(foundstart == TRUE && foundstop == TRUE)
     {
          if((*stop - *start) > MINPERTLEN)
          {
               //if(LOGdata)OutFiledata<<", pertlengthfound ok: " <<
               //(stop-start) << endl;
               return OK;
          }
          else
          {
               i = ZeroOldPertRDPoints(rd, pbrd, (pbrd + SAMPBUFLEN -
               1), start, stop);
               return APDFAIL;
          }
     }
     else
          return APDFAIL;
} //FullPertPresent


/*******************************************************/
int* MaxInt(int* inarray, const int arraysize)
{

      //Find the maximum value of an integer array

     //consts

     //var decs
     int maxvalindex = 0;
     int i = 0;

     //process
     for(i = 0; i<arraysize; i++)
     {
          if(inarray[i] >= inarray[maxvalindex])
               maxvalindex = i;
     }

     //OutFile << "max value index: " << maxvalindex << endl;
     return &inarray[maxvalindex];

} //MaxInt

/*******************************************************/
```

```
float* MaxFloat(float* inarray, const int arraysize)
{
	 //Find the maximum value of a floating point array

	//consts

	//var decs
	int maxvalindex = 0;
	int i = 0;

	//process
	for(i = 0; i<arraysize; i++)
	{
		if(inarray[i] >= inarray[maxvalindex])
			maxvalindex = i;
	}

	//OutFile << "max value index: " << maxvalindex << endl;
	return &inarray[maxvalindex];

} //MaxFloat


/**********************************************************/
int FindMostCommonValue(float* inarray, const int arraysize,
                               const float min, const float max, const
int numbins, float * mcv)
{

	 //Find the most common value in the array
	 //based on bins for floating point values
	 //This is used to set the threshold for
	 //resistance determination

	//Assumes:
	// - positive values only
	// this function executes fastest out of various methods so far

	//consts

	//var decs
	int i = 0;
	int index = 0;
		//index tracker
	int NumInBin[NUMBINS] = {0};
	float interval = 0.0f; //(max - min) / ((float)numbins);
	//bin interval
	float floatval = 0.0f;

	//process
	floatval = *MaxFloat(inarray, arraysize);
	floatval = max > floatval ? max : floatval;

	interval = (floatval - min) / ((float)numbins);

	//count values falling within each bin
	for(i = 0; i < arraysize; i++)
	{
		if(*inarray >= min)
```

```
		{
			index = (int)((*inarray - min + ERRPAD) / interval);
			//OutFile << "Index" << index << " for val: " <<
			//*inarray << "at i = " << i << endl;
			if(index < numbins)
				NumInBin[index]++;
		}
		inarray++;
	}

	//subtract pointers to get index of max value, multiply by
	//interval and add
	//one interval to get upper bound of bin as most common value
	//Old Method for APD-SA: add one interval to bin for threshold
	//*mcv = (interval * (MaxInt(NumInBin, numbins) - NumInBin)) +
	//THRESPAD + min;
	//above was inconsistent with APD100, using exact APD100 method
	//(*1.25)
	//which gives higher threshold, but may keep data more consistent
	//with past collection
	*mcv = 1.25 * ((interval * (MaxInt(NumInBin, numbins) -
	NumInBin)) + min);

	//OutFile << "Index of MCV: " << (MaxInt(NumInBin, numbins) -
	//NumInBin) << ", Num Occurences: "
	//	<< (*MaxInt(NumInBin, numbins))
	//	<< ", interval: " << interval << ", Num at bin 15: "
	//	<< (NumInBin[15]) << ", Num at bin 16: " << (NumInBin[16])
	//<< endl;


	if(1)
		return OK;
	else
		return APDFAIL;

} //FindMostCommonValue

/********************************************************/
void DifferentiateArray(float* inarray, float* outarray, const int
arraysize)
{

	 //Differentiate array starting with first values
	 //to the end

	//consts

	//var decs
	float* pminus1;
	int i = 0;

	//process
	*outarray++ = 0.0f;		//outarray[0] = 0
	pminus1 = inarray++; //pminus = 0, inarray = 1
	for(i=1; i<arraysize-1; i++)
	{
		*outarray++ = *(++inarray) - *pminus1++; //pre-inc inarray
	}
```

```
     *outarray = 0.0f;        //zero last element

     return;
} //DifferentiateArray


/***************************************************/
void CopyCircArrayFrac(Frac16* srcarr, Frac16* destarr, Frac16*
bsrcarr,
                      Frac16* bdestarr, const int length)
{

      //Copy Array for ints from source buffer to
      //destination buffer such that first
      //item in destination buffer is the
      //first item in the time sequence.

     //assumes input source location is incremented to one new
     //location, not yet filled.
     //assumes global arrays
     //this function fills locations but doesn't update pointers

     //var decs
     Frac16* pcurloc;
     int i = 0;

     //process
     pcurloc = srcarr;
     destarr = bdestarr;

     for(i = (srcarr - bsrcarr); i < length; i++)
     //copy last part of rd array
     {
          *destarr++ = *pcurloc++;
     }


     pcurloc = bsrcarr;
     for(i = 0; i < (srcarr - bsrcarr); i++)
               //copy first part of rd array
     {
          *destarr++ = *pcurloc++;
     }

     return;
} //CopyCircArrayFrac



/***************************************************/
void CopyCircArray(float* srcarr, float* destarr, float* bsrcarr,
float* bdestarr, const int length)
{

      //Copy Array from source buffer to
      //destination buffer such that first
      //item in destination buffer is the
      //first item in the time sequence.
```

```
    //assumes input source location is incremented to one new
    //location, not yet filled.
    //assumes global arrays
    //this function fills locations but doesn't update pointers

    //var decs
    float* pcurloc;
    int i = 0;

    //process
    pcurloc = srcarr;
    destarr = bdestarr;

    for(i = (srcarr - bsrcarr); i < length; i++)
    //copy last part of rd array
    {
         *destarr++ = *pcurloc++;
    }


    pcurloc = bsrcarr;
    for(i = 0; i < (srcarr - bsrcarr); i++)
              //copy first part of rd array
    {
         *destarr++ = *pcurloc++;
    }

    return;
} //CopyCircArray


/****************************************************************/
static void DualFilterLoop4Pole100_500(Frac16* valio1, Frac16* valio2,
const int reset)
{

     //This is implementation of an infinite impulse
     //digital filter - Bessel, see section below
     //for poles, cutoff freq.

    static float xv1[NZEROS+1] = {0.0f};
    static float yv1[NPOLES+1] = {0.0f};
    static float xv2[NZEROS+1] = {0.0f};
    static float yv2[NPOLES+1] = {0.0f};
    int ii = 0;

    float _valinout1 = 0.0f;
    float _valinout2 = 0.0f;
    float* valinout1 = &_valinout1;
    float* valinout2 = &_valinout2;


    //*valinout1 = FracToFloat(*valio1);
    //*valinout2 = FracToFloat(*valio2);
    *valinout1 = (float)(*valio1);
    *valinout2 = (float)(*valio2);
```

```
//10 Pole
if(reset == FALSE)
{
     xv1[0] = xv1[1]; xv1[1] = xv1[2]; xv1[2] = xv1[3]; xv1[3] =
     xv1[4]; xv1[4] = xv1[5]; xv1[5] = xv1[6]; xv1[6] = xv1[7];
     xv1[7] = xv1[8]; xv1[8] = xv1[9]; xv1[9] = xv1[10];
   xv1[10] = *valinout1 / GAIN;
   yv1[0] = yv1[1]; yv1[1] = yv1[2]; yv1[2] = yv1[3]; yv1[3] =
     yv1[4]; yv1[4] = yv1[5]; yv1[5] = yv1[6]; yv1[6] = yv1[7]; y
     v1[7] = yv1[8]; yv1[8] = yv1[9]; yv1[9] = yv1[10];
   yv1[10] =   (xv1[0] + xv1[10]) + 10 * (xv1[1] + xv1[9]) + 45 *
     (xv1[2] + xv1[8])
               + 120 * (xv1[3] + xv1[7]) + 210 * (xv1[4] + xv1
     [6]) + 252 * xv1[5]
               + ( -0.0101977570 * yv1[0]) + (  0.1490792186 *
     yv1[1])
               + ( -0.9942499194 * yv1[2]) + (  3.9879880270 *
     yv1[3])
               + (-10.6669162260 * yv1[4]) + ( 19.9083721130 *
     yv1[5])
               + (-26.2993670320 * yv1[6]) + ( 24.3276134390 *
     yv1[7])
               + (-15.1148787730 * yv1[8]) + (  5.7112536127 *
     yv1[9]);
     *valinout1 = yv1[10];

     //process second val
     xv2[0] = xv2[1]; xv2[1] = xv2[2]; xv2[2] = xv2[3]; xv2[3] =
     xv2[4]; xv2[4] = xv2[5]; xv2[5] = xv2[6]; xv2[6] = xv2[7];
     xv2[7] = xv2[8]; xv2[8] = xv2[9]; xv2[9] = xv2[10];
   xv2[10] = *valinout2 / GAIN;
   yv2[0] = yv2[1]; yv2[1] = yv2[2]; yv2[2] = yv2[3]; yv2[3] =
     yv2[4]; yv2[4] = yv2[5]; yv2[5] = yv2[6]; yv2[6] = yv2[7];
     yv2[7] = yv2[8]; yv2[8] = yv2[9]; yv2[9] = yv2[10];
   yv2[10] =   (xv2[0] + xv2[10]) + 10 * (xv2[1] + xv2[9]) + 45 *
     (xv2[2] + xv2[8])
               + 120 * (xv2[3] + xv2[7]) + 210 * (xv2[4] + xv2
     [6]) + 252 * xv2[5]
               + ( -0.0101977570 * yv2[0]) + (  0.1490792186 *
     yv2[1])
               + ( -0.9942499194 * yv2[2]) + (  3.9879880270 *
     yv2[3])
               + (-10.6669162260 * yv2[4]) + ( 19.9083721130 *
     yv2[5])
               + (-26.2993670320 * yv2[6]) + ( 24.3276134390 *
     yv2[7])
               + (-15.1148787730 * yv2[8]) + (  5.7112536127 *
     yv2[9]);
     *valinout2 = yv2[10];
}
```

```
    //NEED TO CONVERT FLOAT BACK TO FRAC16 and WRITE OUT TO REFs
    //*valio1 = FloatToFrac(*valinout1);
    //*valio2 = FloatToFrac(*valinout2);
    *valio1 = (int)(*valinout1);
    *valio2 = (int)(*valinout2);


    if(reset == TRUE)
    {
         for(ii = 0; ii< (NPOLES+1); ii++)
         {
              yv1[ii] = 0.0f;
              yv2[ii] = 0.0f;
         }
         for(ii = 0; ii < (NZEROS + 1); ii++)
         {
              xv1[ii] = 0.0f;
              xv2[ii] = 0.0f;
         }
    }

    return;

} //DualFilterLoop4Pole100_500


/************************************************************/
print_array(int arr[], int length)
{

     //For debugging, print array to screen.

    int i;
    outsci(SCI0DR,"\15\nArray = ");
    for (i=0;i<length;i++)
    {
         outdec(SCI0DR,arr[i]);
    }
    outsci(SCI0DR,"\n");
}


/************************************************************/
void swap (int *a, int *b)
{
     //swap values
    int c = *a;
    *a = *b;
    *b = c;
}

/************************************************************/
/* I/O routines for SCI (i.e. PC hyperterm or minicom communication) */
void inscichar (int *a,char *b)
{
     //Read character in from serial port
     //this is from Peter Gray example
  int status;
```

```
  do status = SCI0SR; while ((status&0x3000)!=0x3000);
  *b = *a;
}
/**********************************************************/
void LCDGoToLineTwo(void)
{

      //Just like the title says.

     outscibyte(SCI0DR, 0xFE); //prepare to receive command
     outscibyte(SCI0DR, 0x47); //go to position
     outscibyte(SCI0DR, 0x01); //column 1 base 1
     outscibyte(SCI0DR, 0x02); //row 2 base 1
}
/**********************************************************/
void LCDClearScreen(void)
{

      //Just like the title says.

     CLS1;
     CLS2;
}
/**********************************************************/
void outscibyte (int *a, const int b)
{

      //send byte out of serial port
      //this is from Peter Gray example

     int status;

     do status = SCI0SR; while ((status&0xC000)!=0xC000);
     *a = b;
     do status = SCI0SR; while ((status&0xC000)!=0xC000);

}
/**********************************************************/
void outscierr(char *b)
{

      //send out error code to LCD display

     LCDClearScreen();
     outsci(SCI0DR, "Err: ");
     //outsci(SCI0DR, "->USB<-");//use this instead since in debugged
     //version,
     //code numbers will signify error type, but an error probably
     //means normal
     //functioning in which a timeout is used to stop an operational
     //mode

     LCDGoToLineTwo();

     outsci(SCI0DR,  b);
}
/**********************************************************/
void outsciusbinfo(char *b)
{
```

```
     LCDClearScreen();
     outsci(SCI0DR, "->USB<-");//use this instead since in debugged
     //version,
     //code numbers will signify error type, but an error probably
     //means normal
     //functioning in which a timeout is used to stop an operational
     //mode

     LCDGoToLineTwo();

     outsci(SCI0DR,  b);
}

/************************************************************/
void outsci (int *a,char *b)
{
      //send out integer or string via serial port
      //this is from Peter Gray example

  unsigned int status;
  while (*b !=0 ) {
    do status = SCI0SR; while ((status&0xC000)!=0xC000);
    *a = *b;
    *b++;
  }
  do status = SCI0SR; while ((status&0xC000)!=0xC000);
}
/************************************************************/
void outscichar (int *a,char *b)
{

      //This is raw send of character out of serial
      //port
      //from Peter Gray example

  int status;
  do status = SCI0SR; while ((status&0xC000)!=0xC000);
  *a = *b;
  do status = SCI0SR; while ((status&0xC000)!=0xC000);
}
/************************************************************/
void outdec (int *chan, int val)
{

      //send decimal integer out via serial port
      //this is from Peter Gray example

  unsigned int u,t;
  char c;
  t = 10000;
  do {
    u = 0;
    while (val>=t) { val -= t; u++; }
    c = '0' + u;
    outscichar (chan,&c);
    t /= 10;
  } while (t>0);
}
/************************************************************/
```

```
void outdecfloat (int *chan, float floatin)
{

	 //Send out floating point value
	 //to serial port

	//var decs
	float rounddivisor = 0.5;
	short digit;
	int digitplace = 0;
	int num10divs = 0;
	float multcounter = 10.0;
	int totdigout = 0;
	int prevaccum = 0;
	int value = 0;
	float copyfloat = 0.0;
	char c;
	int i;

	//debug trial numbers
	copyfloat = floatin;
		//floatin *= 3.999f;
		// to preserve rr value in calling function


	//digit = static_cast<unsigned int>(100.0f * floatin);

	//process

	//** first first check for negative and then abs
	if(floatin < 0.0)
	{
		c = '-';
		outscichar(chan, &c);
		floatin = floatin * -1.0;
	}

	//** first count number power of ten
	while((int)(floatin / multcounter) != 0)
	{
		num10divs++;
		multcounter *= 10.0;
	}
	//** loop to output each digit
	totdigout = num10divs + 1 + DIGOUTPRECIS;
	for(i = 0; i < totdigout; i++)
	//to prep for rounding:
		rounddivisor /= 10.0;
	// - create 5 digit to add to LSDig
	floatin /= multcounter;
	//start with decimal at beginning
	floatin += rounddivisor;
	//add the 5 to LSDig
	multcounter = 10.0;
	//reset multcounter
	//OutFile << "floatin: " << floatin << endl;

	for(i = 0; i < totdigout; i++) //just changed this to < from <=
	{
```

```
            //OutFile << "totdigout: " << totdigout << endl;
            prevaccum *= 10;
            value = (long)(floatin * multcounter);
            digit = (short)(value - prevaccum);
            //OutFile << "floatin * multcounter: " << (value)
            //<< ", (short)(floatin * multcounter): " << digit << endl;
            multcounter *= 10.0;
            prevaccum = value;

            //output'digit' value here
            //for decimal place on MAX and LED chips use
            //digitplace == DPDIG
            digitplace = totdigout - i;
            if(digitplace == (DPDIG-1)) //might need decimal place
            {
                  c = '.';
                  outscichar(chan, &c);
            }
            c = '0'+digit;
            outscichar (chan,&c);

            /*
            if(USEMAX7221)
            {
                  digitplace = totdigout - i;
                  if(digitplace == 4 && digit == 0)digit = 0x0f;
                  //if MSD is zero, blank it
                  if(digitplace == DPDIG)digit |= 0x80;
                  //turn on dec pt if necess
                  WriteOutDigitMAX7221(digit, digitplace);
            }
            else
            {
                  digitplace = totdigout - i - 1;
                  WriteOutDigit4511(digit, digitplace);
            }
            //LoopWait(1000);
            */
      }

      floatin = copyfloat;
      //if debug, return original ref value to original state

}
/******************************************************/
void outscihex (int *a,int b)
{

       //send hex value out of serial port
       //this is from Peter Gray's example

  int  c;
  int  r;
  char t[5];
  c = 3;
  while (c>=0) {
    r = b&0x000F;
    if (r < 10)
      t[c] = '0'+r;
```

```
    else
      t[c] = 'A'+((r)-10);
    b >>= 4;
    b = b&0x0FFF;
    c--;
  }
  t[4] = 0;
  outsci (a,t);
}



/****************************************************************/
int SendAckToUSB(void)
{

     //send ack byte to USB portA via porta 803

     int i = 0x00;
     int wdog = 0x0000;

     PADDR = DT_USB_DDR; //set pins to out
     USB_DSP_ACK;         //send ACK byte to USB
     DR_to_USB; //interrupt USB to receive data
     do{
          i=USB_ACK;
          wdog++;
     } while(i==0 && wdog<0xfff);
//wait for USB to say ok I heard your ACK
//at 80MHz, 10 instructions per loop
//fff=4095 is about 0.8 ms
//reset the data ready flag - hopefully this will avoid unwanted and
//further INTs
//on the USB - though I don't know if this is true...question for EZ-
//USB
//if INT6 (edge-sensitive) is latched, but not cleared when the int
//flag
//is cleared from within the ISR, will another INT6 be flagged? even
//though
//there was not an actual transition? we shall see...
//NDR_to_USB;
//yup, it looks as if this is the case...how annoying, though
///////should
//have written the NDR anyway to be sure, ya know...anyway, at least
//now I know.



     //if not an overflow of watchdog, then continue ack cycle
     if(wdog<0xfff)
     {
          wdog=0; //reset wdog for next wait...

          do {
               i=USB_ACK;
               wdog++;
          } while(i!=0 && wdog<0xfff);
          //wait for USB to finish saying ok
```

```
	}

	if(wdog>=0xfff)
	{
		return APDFAIL;
		if(DEBUG)outsci(SCI0DR, "AF:ACK2USB");
		if(USELCD)outscierr("A2Ux");
	} else
		return OK;

}

/************************************************/
#pragma interrupt saveall
void isrIRQB ()
{

	 //ISRB is the USB control interrupt
	 //IRQ B is triggered and then data is read
	 //from port to determine type of action to
	 //complete

	//Some information for you:
	//IRQB is second highest priority interrupt and is en/dis
	//by IPR bit 4 - it is second in priority to IRQA and
	//then to the level 1 non-maskable interrupts
	//thus you cannot watchdog a routine called from within
	//it...because the IRQ will never be interrupted by
	//lower priority interrupts - even if set to highest
	//priority as a Ch6 assignable peripheral interrupt
	//level....thus, just use a loop counter and break on failure
	//of a routine, mkay? - you got the time it appears as
	//all execution is pretty fast, and hopefully the resources
	//for a non-static variable, mkay?


	//This (IRQB) is a USB command assertion

	static j=0;
	int i;
	int porta = 0x00;
	int success = 0;

	if(DEBUG)
		outsci(SCI0DR, "Qs ");

	TMRB0_CTRL = 0x0000;	//Turn off any running timer

	duty = 0x0000;		//Turn off motor
	SetPWMDuty(&duty);


	IPR_W = (IPR_W & ~0x0010);// & ~0x0002); //disable IRQA
	DESELECTADS;		//make sure no ADS selected
	NDR_to_USB;			//no data to USB (don't int it)

	PADDR = DF_USB_DDR;  //read PA to check for appropriate USB byte
	porta = PADR;
```

```
if(DEBUG){j++;} //number of times this called for debug stuff

if(!(mode & USBATTACHMASK))
{    //USB not yet attached
     if(porta==USB_HELLO)
     //proper byte from USB is preset to attach
     {
          success = SendAckToUSB();
          //First send ACK byte to USB

          //if timeout or no transfer
          if(success==APDFAIL)
               goto end_of_irqb;

          mode = (mode | USBATTACHMASK);
          //set USB_ATTACHED bit

          if(DEBUG){
               LCDClearScreen();
               outsci(SCI0DR, "UC ");}
          if(USELCD){
               //LCDClearScreen();
               //outsci(SCI0DR, "->USB<-");
               //LCDGoToLineTwo();
               //outsci(SCI0DR, "Connect");
               //outsci(SCI0DR, "01"); //just use codes
               outsciusbinfo("01");
          }



          mode |= STOPMODE;
          //Set mode for poll loop

          goto end_of_irqb;
          //Just in case, jump to end of this

     } else { //not proper byte to connect
          LCDClearScreen();
          if(DEBUG){
               outsci(SCI0DR, "U?");
               outscihex(SCI0DR, porta);}
          if(USELCD){
               //outscierr("U?");
               outsciusbinfo("02");
          }

          goto end_of_irqb;
     }
} else { //USB already attached
     //Select command from the USB
     //outsci(SCI0DR, "UA ");
     switch (porta)
     {
          case USB_REQUEST_VARIABLES:

               g_process=0;
               TMRA0_CNTR=0;
               STARTCHECKTIME;
```

```
        success = SendAckToUSB();
        //send ACK Byte to USB port A

        g_process = TMRA0_CNTR;
        STOPCHECKTIME;
        TMRA0_CNTR=0;
        if(DEBUG){
             outscibyte(SCI0DR, 0xFE);
             outscibyte(SCI0DR, 0x58);
             outdec(SCI0DR, g_process);
        }
        g_process=0;

        //if timeout or no transfer
        if(success==APDFAIL)
             goto end_of_irqb;

        if(DEBUG){
             LCDClearScreen();
             outsci(SCI0DR, "UR ");}

        if(USELCD){
             outsciusbinfo("03");}

        success = SendVars();
        //send over all the variables

        //if timeout or no transfer
        if(success==APDFAIL)
             goto end_of_irqb;

        break;

   case USB_HELLO:
        //already connected
        success = SendAckToUSB();

        //if timeout or no transfer
        if(success==APDFAIL)
             goto end_of_irqb;

        //connected bit is already set so leave it alone
        if(DEBUG){
             LCDClearScreen();
             outsci(SCI0DR, "URC ");}

        if(USELCD){
             outsciusbinfo("04");}

        //just in case in run mode, stream mode etc.
        //set this to allow re-connect to stop
        //any transfers etc.
        mode |= STOPMODE;
        break;

   case USB_STREAM_ADC:
        success = SendAckToUSB();
```

```
        //if timeout or no transfer
        if(success==APDFAIL)
             goto end_of_irqb;

        //now wait for EZUSB to finish doing its
        //comm with host
        LoopSWait(0xffff);

        if(DEBUG){
             LCDClearScreen();
             outsci(SCI0DR, "US ");}

        if(USELCD){
             outsciusbinfo("05");}

        mode |= STREAMMODE;
        break;

   case USB_RUN:
        success = SendAckToUSB();

        //if timeout or no transfer
        if(success==APDFAIL)
             goto end_of_irqb;

        //now wait for EZUSB to finish doing its
        //comm with host
        LoopSWait(0xffff);

        if(DEBUG){
             LCDClearScreen();
             outsci(SCI0DR, "UN ");}

        if(USELCD){
             outsciusbinfo("06");}

        mode |= USBRUNMODE;
        break;

   case USB_STOP:
        success = SendAckToUSB();

        //if timeout or if no transfer
        if(success==APDFAIL)
             goto end_of_irqb;

        if(DEBUG){
             LCDClearScreen();
             outsci(SCI0DR, "UX ");}

        if(USELCD){
             outsciusbinfo("07");}

        mode |= STOPMODE;
        break;

   case USB_WRITE_VARS:
        success = SendAckToUSB();
```

```
                //if timeout or if no transfer
                if(success==APDFAIL)
                     goto end_of_irqb;

                if(DEBUG){
                     LCDClearScreen();
                     outsci(SCI0DR, "UW ");}

                if(USELCD){
                     outsciusbinfo("08");}

                WriteFlashConstants();

                break;

           case USB_GET_HOST_VARS:
                success = SendAckToUSB();

                //if timeout or no transfer
                if(success==APDFAIL)
                     goto end_of_irqb;

                if(DEBUG){
                     LCDClearScreen();
                     outsci(SCI0DR, "UG ");}

                if(USELCD){
                     outsciusbinfo("09");}

                success = GetHostVars();

                //if timeout or no transfer
                if(success==APDFAIL)
                     goto end_of_irqb;

                WriteFlashConstants();

                break;

           default:
                if(DEBUG){
                     outsci(SCI0DR, "UU ");
                     outscihex(SCI0DR, porta);}
                //if(USELCD){
                //   outscierr("UU");}
                if(USELCD){
                     outsciusbinfo("10");}

                break;
      }

}


//change run mode to slave mode? the config vars to the Main PC

//if(mode==STOPMODE){};
//this shouldn't really ever happen
```

```
	if(mode==NOCALLMODE){};
	//after stuff has already been cleared in the main loop

	//if(mode==RUNMODE)mode=STOPMODE;

end_of_irqb:
if(DEBUG){
	outsci(SCI0DR, "Q#");
	outscihex(SCI0DR, j);}

//if timeout or no transfer
if(success==APDFAIL)
{
	if(DEBUG)outsci(SCI0DR, "AF");
	if(USELCD)outscierr("AF");
	//maybe consider putting here a reset to RUNMODE if failed
	//for example if failure due to disconnect...
	//OR could just leave it so that to resume in the case of
	//disconnect, just
	//recycle power....yes, yes.

}
IPR_W = (IPR_W | 0x0010); //0x0002); //re-enable IRQB

}


/*******************************************************************/
#pragma interrupt saveall
void isrTimerB0Compare ()
{


	 //Timer interrupt routine for TIMERB0

	/*short first = 0x0000;
	short second = 0x0000;
	int val = 0x0000;
	static int cntr = 0;
	static int cntr2 = 0;
	static float flt_val = 0.0;
	*/

	//process

	(*g_pfcn)();




	/* clear compare status flags */
	asm (bfclr #$8000,X:TMRB0_SCR_R);
	//another timer compare int shouldn't fire
	//until after this bit is cleared.

}
```

```
;LISTING D.4.  56803_vector_pROM.asm

;----------------------------------------------------------------------
;  For APD-SA USB 56803 chip
;  The framework was provided in the Motorola embedded 56800
;  development package.
;
;  Isr Routines that were added for APD.
;    ISRs: TimerB0 and IRQB
;
;  N. Silverman
;
;  -------------------------------------------------------
;  Metrowerks Embedded Runtime Support
;
;    56803_vector_pROM.asm
;
;    sample code
;    Metrowerks, a Motorola Company
;
;
;
;    Routines
;    --------
;
;    These are the interrupt vectors for the DSP56803
;
;
;----------------------------------------------------------------------

     section rtlib
     org   p:

M56803_intRoutine:
     nop
     rti

M56803_illegal:
     debug      ; illegal instruction interrupt ($04)
     nop
     nop

M56803_HWSOverflow:
     debug      ; hardware stack overflow interrupt ($08)
     nop
     nop

M56803_PLL:
     debug      ; PLL lost of lock interrupt ($28)
     nop
     nop

M56803_intDef:
     nop
     rti
    endsec

     section interrupt_vectors_mirror
     org   p:
```

```
 jmp Finit_M56803_        ; reset                     ($00)
 jmp M56803_intRoutine    ; COP Watchdog reset        ($02)
endsec

 section interrupt_vectors
 org   p:

 jmp M56803_intRoutine    ; reserved                  ($04)
 jmp M56803_illegal       ; illegal instruction       ($06)
 jmp M56803_intRoutine    ; Software interrupt        ($08)
 jmp M56803_HWSOverflow   ; hardware stack overflow   ($0A)
 jmp M56803_intRoutine    ; OnCE Trap                    ($0C)
 jmp M56803_intRoutine    ; reserved                  ($0E)
 jmp M56803_intRoutine    ; external interrupt A      ($10)
 jmp FisrIRQB             ; external interrupt B      ($12)
 jmp M56803_intRoutine    ; reserved                     ($14)
 jmp M56803_intRoutine    ; boot flash interface      ($16)
 jmp M56803_intRoutine    ; program flash interface   ($18)
 jmp M56803_intRoutine    ; data flash interface      ($1A)
 jmp M56803_intRoutine    ; mscan transmitter ready   ($1C)
 jmp M56803_intRoutine    ; mscan receiver full       ($1E)
 jmp M56803_intRoutine    ; mscan error               ($20)
 jmp M56803_intRoutine    ; mscan wakeup              ($22)
 jmp M56803_intRoutine    ; program flash interface 2($24)
 jmp M56803_intRoutine    ; GPIO E                    ($26)
 jmp M56803_intRoutine    ; GPIO D                    ($28)
 jmp M56803_intRoutine    ; reserved                  ($2A)
 jmp M56803_intRoutine    ; GPIO B                    ($2C)
 jmp M56803_intRoutine    ; GPIO A                    ($2E)
 jmp M56803_intRoutine    ; SPI transmitted empty     ($30)
 jmp M56803_intRoutine    ; SPI receiver full/error   ($32)
 jmp M56803_intRoutine    ; Quad decoder #1 home sw   ($34)
 jmp M56803_intRoutine    ; Quad decoder #1 idx pulse($36)
 jmp M56803_intRoutine    ; Quad decoder #0 home sw   ($38)
 jmp M56803_intRoutine    ; Quad decoder #0 idx pulse($3A)
 jmp M56803_intRoutine   ; Timer D Channel 0        ($3C)
 jmp M56803_intRoutine   ; Timer D Channel 1        ($3E)
 jmp M56803_intRoutine    ; Timer D Channel 2         ($40)
 jmp M56803_intRoutine    ; Timer D Channel 3         ($42)
 jmp M56803_intRoutine    ; Timer C Channel 0         ($44)
 jmp M56803_intRoutine    ; Timer C Channel 1         ($46)
 jmp M56803_intRoutine    ; Timer C Channel 2         ($48)
 jmp M56803_intRoutine    ; Timer C Channel 3         ($4a)
 jmp FisrTimerB0Compare   ; Timer B Channel 0         ($4c)
 jmp M56803_intRoutine    ; Timer B Channel 1         ($4e)
 jmp M56803_intRoutine    ; Timer B Channel 2         ($50)
 jmp M56803_intRoutine    ; Timer B Channel 3         ($52)
 jmp M56803_intRoutine    ; Timer A Channel 0         ($54)
 jmp M56803_intRoutine    ; Timer A Channel 1         ($56)
 jmp M56803_intRoutine    ; Timer A Channel 2         ($58)
 jmp M56803_intRoutine    ; Timer A Channel 3         ($5a)
 jmp M56803_intRoutine    ; SCI #1 Transmit complete ($5c)
 jmp M56803_intRoutine    ; SCI #1 transmitter ready ($5e)
 jmp M56803_intRoutine    ; SCI #1 receiver error    ($60)
 jmp M56803_intRoutine    ; SCI #1 receiver full     ($62)
 jmp M56803_intRoutine    ; SCI #0 Transmit complete ($64)
 jmp M56803_intRoutine    ; SCI #0 transmitter ready ($66)
 jmp M56803_intRoutine    ; SCI #0 receiver error    ($68)
```

```
        jmp M56803_intRoutine    ; SCI #0 receiver full     ($6a)
        jmp M56803_intRoutine    ; ADC B Conversion complete($6c)
        jmp M56803_intRoutine    ; ADC A Conversion complete($6e)
        jmp M56803_intRoutine    ; ADC B zero crossing/error($70)
        jmp M56803_intRoutine    ; ADC A zero crossing/error($72)
        jmp M56803_intRoutine    ; Reload PWM B              ($74)
        jmp M56803_intRoutine    ; Reload PWM A              ($76)
        jmp M56803_intRoutine    ; PWM B Fault                     ($78)
        jmp M56803_intRoutine    ; PWM A Fault                     ($7a)
        jmp M56803_PLL           ; PLL loss of lock                ($7c)
        jmp M56803_intRoutine    ; low voltage detector      ($7e)

        endsec



M56803_OMRSetting equ $0103

        section         rtlib

        org   x:
      global FM56803_int_Addr
FM56803_int_Addr dc   M56803_intDef  ; Address of the unhandled
exception

M56803_argc equu  0
        global FM56803_argv
        global FM56803_arge
FM56803_argv:
FM56803_arge:    dc   0


        endsec
        end
```

APPENDIX E: USB FIRMWARE CODE

Code Listings:

- Listing E.1. apdusb.h – header file including custom APD-SA USB commands
- Listing E.2. DSCR.A51 – descriptor table for the EZUSB firmware that indicates to the host the device identity and requirements of the APD-SA USB on attachment
- Listing E.3. timer0.c – timer routines for watchdog and data transmission timeout routines
- Listing E.4. ezmouse.c – main code for the EZUSB firmware for the APD-SA USB

```
//LISTING E.1.  apdusb.h

/*******************************************

 apdusb.h header file for inclusion in
  ezmouse.c and timer0.c

 Macro Definitions for EZUSB APD-SA USB firmware:
  - identifying control bytes in data transmission.
  - bitwise register manipulation
  - data formatting

 Macros defined here are mirrored almost entirely in the
 APD200 host PC software and to some extent in the DSP
 firmware where needed.

 Indented comments refer to the macro definition above the
 comment.

 N. Silverman 2004 MS Thesis

*******************************************/


#define USB_CONNECT_TO_DSP                0x01
#define USB_UNRECOGNIZED_COMMAND          0xff
#define USB_DSP_NO_ACK                    0xfe
#define USB_DSP_ACK                       0xbb
#define USB_REQUEST_VARIABLES             0x02
#define USB_END_OF_VARIABLE_TRANSFER      0xfd
#define USB_HELLO                         0xb0
#define USB_NUMVARS                       2
#define USB_BYTES_PER_VAR                 6
     //bytes per variable - sent as char
#define USB_MAX_BYTES                     64
     //max bytes in buffer
#define USB_EN_INT6                       EIE|=0x10
     //enable INT6
#define USB_DIS_INT6                      EIE&=~0x10
     //disable INT6
#define USB_EN_USBINT                     EIE|=0x01
     //enable USB INTs
#define USB_DIS_USBINT                    EIE&=~0x01
     //disable USB INTs
#define USB_STREAM_ADC                    0x03
     //stream ADC values request
#define USB_STALL_FAILURE                 0xfc
     //USB unit stalled and had timer interrupt - watchdog
#define USB_STOP                          0xfb
     //tell DSP to stop whatever is continuously occurring...
     //(or anything)
#define USB_GET_HOST_VARS                 0x05
     //host computer to send variables to apd for storage
#define USB_RUN                           0x06
     //run mode on the APD with streaming of data -
     //rr, press, flow vals
#define USB_RUN_FULL                      0x07
     //run mode with stream of full pert info data,
     //without each ADC value
```

```
;; LISTING E.2.  DSCR.A51
;;-----------------------------------------------------------------------
;;   dscr.a51
;;
;;   For APD-SA USB, the EZUSB firmware is intended to have the
;;   device enumerate as an HID mouse device
;;
;;   This descriptor table is passed to the host and lets the host
;;   know what type of device it is and what its requirements for
;;   data transmission bandwidth and needed endpoints are.
;;
;;   This file was constructed from various samples provided by
;;   Cypress Semiconductor, Inc. (2001)
;;
;;   Adaptation was however extensive.
;;
;;   The adaptation here combines USB HID mouse identification
;;   but includes the specification of additional endpoints as
;;   interrupt endpoints.
;;
;;   Jan Axelson (USB Complete, 2001)
;;   and John Hyde (http://www.usb-by-example.com)
;;   and the USB Consortium (http://www.usb.org) provide
;;   very useful information that describe the details of the
;;   sections of the descriptor table.  In particular, usb.org
;;   provides the specification documents for HID and USB 1.1
;;   as well as 2.0 of course.
;;
;;   Adaptation: N. Silverman 2004 MS Thesis
;;
;;   Note that indented comments typically follow the line to which
;;   they refer.
;;
;;   Contents:  This file contains descriptor data for
;;   sample mouse descriptor tables.
;;
;;   Original sample descriptor:
;;   Copyright (c) 2001 Cypress Semiconductor, Inc.
;;   All rights reserved
;;-----------------------------------------------------------------------

DSCR_DEVICEequ   1     ;; Descriptor type: Device
DSCR_CONFIGequ   2     ;; Descriptor type: Configuration
DSCR_STRINGequ   3     ;; Descriptor type: String
DSCR_INTRFCequ   4     ;; Descriptor type: Interface
DSCR_ENDPNTequ   5     ;; Descriptor type: Endpoint

ET_CONTROL equ   0     ;; Endpoint type: Control
ET_ISO           equ   1     ;; Endpoint type: Isochronous
ET_BULK          equ   2     ;; Endpoint type: Bulk
ET_INT           equ   3     ;; Endpoint type: Interrupt

public           DeviceDscr, ConfigDscr, StringDscr, UserDscr, HIDDscr,
ReportDscr, ReportDscrEnd

DSCR SEGMENT     CODE

;;-----------------------------------------------------------------------
```

```
;; Global Variables
;;-----------------------------------------------------------------------
;; Note: This segment must be located in on-part memory.
           rseg DSCR
;; locate the descriptor table anywhere below 8K
DeviceDscr:db   deviceDscrEnd-DeviceDscr       ;; Descriptor length
           db   DSCR_DEVICE    ;; Decriptor type
           dw   0001H      ;; Specification Version (BCD)
           db   00H        ;; Device class
           db   00H        ;; Device sub-class
           db   00H        ;; Device sub-sub-class
           db   64         ;; Maximum packet size
           /*dw 4705H      ;; Vendor ID
           dw   0210H      ;; Product ID
           dw   0100H      ;; Product version ID
           db   1          ;; Manufacturer string index
           db   2          ;; Product string index
           db   0          ;; Serial number string index
           db   1          ;; Numder of configurations
           */
           dw   4705H      ;; Vendor ID
                           ;;this shows up in windows as 0547 NS
           dw   2810H      ;;0210H ;;2810H
                           ;; Product ID - set to default example ID
                           ;;this shows up in windows as 1028 NS
           dw   0100H      ;; Product version ID
                           ;;this shows up in windows as 0001 NS
           db   0          ;; Manufacturer string index
           db   0          ;; Product string index
           db   0          ;; Serial number string index
           db   1          ;; Number of configurations
deviceDscrEnd:

ConfigDscr:db   ConfigDscrEnd-ConfigDscr
                           ;; Descriptor length
           db   DSCR_CONFIG    ;; Descriptor type
           db   StringDscr-ConfigDscr
                           ;; Configuration + End Points length (LSB)
           db   (StringDscr-ConfigDscr)/256
                           ;; Configuration + End Points length (MSB)
           db   1          ;; Number of interfaces
           db   1          ;; Interface number
           db   0          ;; Configuration string
           db   10100000b
                ;; Attributes (b7 - buspwr, b6 - selfpwr, b5 - rwu)
           db   0          ;; Power requirement (div 2 ma)
ConfigDscrEnd:

IntrfcDscr:
           db   IntrfcDscrEnd-IntrfcDscr       ;; Descriptor length
           db   DSCR_INTRFC    ;; Descriptor type
           db   0          ;; Zero-based index of this interface
           db   0          ;; Alternate setting
           db   2          ;; Number of end points
           db   03H        ;; Interface class (HID)
                           ;;ffH for ep_pair NS
           db   00h        ;;01H
                           ;; Boot Interface sub class
                           ;;00h for ep_pair NS
```

```
            db   00h        ;; 02H
                            ;; Interface sub sub class (Mouse)
                            ;;00H for ep_pair NS
            db   0          ;; Interface descriptor string index
IntrfcDscrEnd:

;; this works NS
HIDDscr:
            db   09h        ; length
            db   21h        ; type: HID
            db   10h,01h        ; release: HID class rev 1.1
            db   00h        ; country code (none)
            db   01h        ; number of HID class descriptors to follow
            db   22h        ; report descriptor type (HID)
            db   (ReportDscrEnd-ReportDscr) ; length of HID descriptor
            db   00h
HIDDscrEnd:


EpInDscr:
            db   EpInDscrEnd-EpInDscr            ;; Descriptor length
            db   DSCR_ENDPNT         ;; Descriptor type
            db   82H                 ;; Endpoint number, and direction
            db   ET_INT              ;; Endpoint type
            db   40H                 ;; Maximum packet size (LSB)
            db   00H                 ;; Max packet size (MSB)
            db   01H                 ;; Polling interval
                                     ;;00H for ep_pair -
                                     ;;changed to 1 ms
                                     ;;for EP IN (ns)
EpInDscrEnd:

                            ;;from ep_pair
EpOutDscr:
            db   EpOutDscrEnd-EpOutDscr          ;; Descriptor length
            db   DSCR_ENDPNT    ;; Descriptor type
            db   02H        ;; Endpoint number, and direction
            db   ET_INT         ;; Endpoint type
            db   40H        ;; Maximun packet size (LSB)
            db   00H        ;; Max packect size (MSB)
            db   00H        ;; Polling interval
EpOutDscrEnd:

/*
ReportDscr:
     db 09h, 01h
     db 06h,00h,ffh
     db 1bh,01h,00h,00h,ffh
     db 29h,01h
     db a1h,00h
     db 15h, 81h
     db 26h, 80h, 00h
     db 75h, 08h
     db 95h, 02h
     db c0h
EndReportDscr:
*/

;; this works NS
```

```
ReportDscr:
     db 05h, 01h     ;     Usage Page (Generic Desktop),
     ;;db 05h, A0h, FFh    ;; Usage Page (vendor defined)
     db 09h, 02h      ;; Usage (vendor defined)

     db 0A1h, 01h          ;; Collection (Application)
     db 09h, 01h      ;; Usage (vendor defined)

     ;;Input report
     db 09h, 01h      ;; Usage (vendor defined)
     ;;db 15h, 80h         ;; Log Min (-127)
     ;;db 25h, 7Fh         ;; Logical Max (128)
     db 75h, 08h      ;; Report Size (8) (bits)
     db 95h, 40h      ;; Report Count (2) (fields)  now64
     db 81h, 02h      ;; Input (Data, Variable, Absolute)

     ;;Output report
     db 09h, 01h      ;; Usage (vendor defined)
     ;;db 15h, 80h         ;; Log Min (-127)
     ;;db 25h, 7Fh         ;; Logical Max (128)
     db 75h, 08h      ;; Report Size (8) (bits)
     db 95h, 40h      ;; Report Count (2) (fields) now64
     db 91h, 02h      ;; Output (Data, Variable, Absolute)

     db 0C0h               ;; End Collection
ReportDscrEnd:

StringDscr:
StringDscr0:
           db   StringDscr0End-StringDscr0            ;; String
descriptor length
           db   DSCR_STRING
           db   09H,04H
StringDscr0End:

StringDscr1:
           db   StringDscr1End-StringDscr1            ;; String
descriptor length
           db   DSCR_STRING
           db   'U',00                          ;;C
           db   'M',00                          ;;Y
           db   'C',00                          ;;P
           db   'P',00                          ;;R
           db   'e',00
           db   's',00
           db   's',00
StringDscr1End:

StringDscr2:
           db   StringDscr2End-StringDscr2            ;; Descriptor
length
           db   DSCR_STRING
           db   'E',00
           db   'Z',00
           db   '-',00
           db   'M',00
           db   'o',00
           db   'u',00
           db   's',00
```

```
            db    'e',00
StringDscr2End:

UserDscr:
            dw    0000H
            end
```

```
// LISTING E.3. timer0.c
//-------------------------------------------------------------------
//   timer0.c
//
//   This code contains functions that set, reset and otherwise
//   control a timer on the 8051 portion of the EZUSB chip.  It is
//   based around a timer interrupt function, called when a
//   compare occurs that triggers an interrupt.
//
//   The routines set the timer count speed, and cascade two
//   counters for higher count.  The interrupt is used as a
//   watchdog timer in the APD-SA USB EZUSB firmware.  It can
//   be used to reset after any lockups or as a tool to signal
//   communication timeouts that are intended as part of the
//   communication protocol.
//
//   Beware, the timer interrupt is masked in other portions
//   of the code in order to protect the integrity of other data
//   transmission and thus may not serve as an appropriate watchdog
//   timer.
//
//   Indented comments typically follow the line to which they refer.
//
//
//
//
//
//   Note that serial out port was used for debug data but
//   most of this code has been commented out here.
//
//   This code was adapted (extensively) from an example of timer
//   use on the Cypress EZUSB development platform that was intended
//   to show timed counting on a LED digit display. See below for
//   for original documentation for that example.
//
//   N. Silverman 2004 for MS Thesis
//
//-------------------------------------------------------------------
//
//   Contents:  Programmable timer interrupt that controls step
//   interval (ranging from 1-5 s)
//   of LED display from 0-9, using endpoint control from the
//   Control Panel.
//   Length Field: Controls how many times the 0-9 count will loop.
//   Hex Bytes Field:    Controls how fast the count steps
//                             1) 01h -> 1s
//                             2) 02h -> 2s
//                             3) 03h -> 3s
//                             4) 04h -> 4s
//                             5) 05h -> 5s
//
//   Author:          Zin Thein Kyaw
//
//   Copyright (c) 2000 Cypress Semiconductor. All rights reserved
//--------------------------------------------------------------------

#include <ezusb.h>
#include <ezregs.h>
#include <apdusb.h> //my header mano - for the APD-SA USB
```

```
extern BYTE     gotfreshdata;
extern BYTEreceivedatablock;
     //so that INT6 isn't re-enables during IN2EP INT
extern BYTEincount;
extern BYTEoutcount;
extern BYTEirqcalled;

extern void TD_Init(void);
void HitWatchdog(void);
extern void outsci(char *b);

//extern void Dummy(void);

// 10ms interrupt
#define TIMER0_COUNT 0x0001
     //this inits so full count is int = 0.0328 seconds
     //maybe not 0x0000 this might trigger INT?????
     //TIMER0_COUNT 0xB1E0
     // 10000h - ((24,000,000 Hz / (12 * 100))

     //as 0xffff - 0xb1e0 = 19999 * (1/2000000) = 10 ms
     // EZ-USB 8051 runs on either a 4-clock bus cycle
     // or the traditional 12-clock bus cycle
static unsigned timer0_tick;
// timer tick variable


//----------------------------------------------------------------------
// Timer Interrupt
// This function is an interrupt service routine for Timer 0.
// It should never
// be called by a C or assembly function. It will be executed
// automatically
// when Timer 0 overflows.
// "interrupt 1" tells the compiler to look for this ISR at
// address 000Bh
// "using 1" tells the compiler to use register bank 1

void timer0 (void) interrupt 1 using 1
{
     // Stop Timer 0, adjust the Timer 0 counter so that
     //we get another in 10ms,
     // and restart the timer.

     static int j=0x00;

     EA=0;
     //disable all interrupts...to service this without
     //interruption (would only be from an INT6)

     TR0 = 0;
     // stop timer

     //outsci("t ");
     //**MINE - I guess do some stuff here, eh?
     //Let's arm endpoints in the case of application (host)
     //stall on waiting for locked up
     //EZUSB
```

```
//you (niz) might want to consider dumping some debug data
//into this buffer, e.g. counts etc.

//since 0 to 0xffff = 0.0328 second, do 35 hits ~=
//1 sec to trigger actual something or other.
//0x7f is about 4 seconds

if(timer0_tick >= 0x7f)
{

      //Dummy();
      outsci("o ");

      if(j>(USB_MAX_BYTES-1)){j=0x00;}
      //reset j if it's larger than buffer EP size

      //setup some type of "oops I stalled" code...
      IN2BUF[j] = USB_STALL_FAILURE;
      IN2BUF[j+1]= timer0_tick;

      //just place stall code in next spot
      j+=2;

      //reset counter
      timer0_tick = 0;

      //clear pending interrupts
      IN07IRQ |= 0x04;
      //clear USB EP2IN  int - cleared by writing a 1 to it
      OUT07IRQ |= 0x04;
      //clear USB EP2OUT int -  cleared by writing a 1 to it
      //USBIRQ=0x00;
      EXIF &= ~0x10;
      //clear USBINTs
      EICON &= ~0x08; //|= 0x08;
      //this one is not reset by writing to it...annoying.
      //clear INT6

      //enable interrupts and start over -
      //nope this is not so good really...
      //TD_Init(); <---------------'

      OEA =0x00;      // init as input...
      OUTA = 0x00;
      // Initialize PORTA to all LOW if used as an output

      USB_DIS_INT6;                   // disable INT6

      OUTB |= 0x20;
      // IRQ is active low, so initialize the
      //803 IRQA = EZ PB5 as high
      // make sure you don't do = 0x20; here cuz it'll
      //drop the whole register and thus give an
      // "all done getting that port data buddy"
      // to that speedy little dsp, ya know?
      gotfreshdata = 0x00;
      receivedatablock = 0;
      //so that INT6 isn't re-enables during IN2EP INT
      //incount=0;
```

```
        //outcount=0;
        irqcalled=0;

        //reset timer
        TL0 = TL0 + (TIMER0_COUNT & 0x00FF);
        TH0 = TH0 + (TIMER0_COUNT >> 8);

        // Increment the timer tick. This interrupt
        // should occur approximately
        // every 10ms. So, the resolution of the timer
        //will be 100Hz not
        // including interrupt latency. - used to be -
        //this comment is left over from sample with old
        //COUNT reset value
        timer0_tick++;

        TR0 = 1;// start Timer 0

        EA=1; //re-enable interrupts - really just allow
        //future USBINTs at this point...

        //enable USB INTs for EP servicing
        USB_EN_USBINT;

        //dump buffer and enable OUT for further future
        //command - this should happen anyway with reset
        EPIO[OUT2BUF_ID].bytes=0;
        //out first to allow new commands and because
        //IN can be picked up whenever
        EPIO[IN2BUF_ID].bytes=64;

        //reset- yeah right, that's not gonna work -
        //it's host access only....how can it run after
        //resetting itself....?
        //CPUCS |= 0x01; //hold in reset
        //CPUCS=0;       //release

  } else {



        //timer should be set as high priority as well - will
        //be masked during the INT6 (since it is set as high
        //priority, but
        //will not be masked during USB transfer stalls, i.e. in
        //the middle of a USB interrupt and waiting for return
        //from the dsp that doesn't happen because its stuck

        //perhaps this routine might return the address of the
        //current instruction, or some general failure due to stall
        //code....yes, let's try that for now...


        TL0 = TL0 + (TIMER0_COUNT & 0x00FF);
        TH0 = TH0 + (TIMER0_COUNT >> 8);

        // Increment the timer tick. This interrupt should occur
        //approximately
        // every 10ms. So, the resolution of the timer will be
```

```
            //100Hz not
            // including interrupt latency.

            timer0_tick++;

            TR0 = 1;
                       // start Timer 0

            EA=1; //re-enable interrupts

      }
}

void HitWatchdog(void)
{

      EA=0;
      TR0 = 0;
            // stop timer

            //reset the tick counter...holy ** man,
            //I can't believe I forgot this...
      timer0_tick = 0;

      TL0 = TL0 + (TIMER0_COUNT & 0x00FF);
      TH0 = TH0 + (TIMER0_COUNT >> 8);

      TR0 = 1;
                 // start Timer 0
      //outsci("h ");
            //ok, this is being reached as it's supposed
            //to be...lots of h's
      EA=1;
}

void timer0_init (void);

// This function enables Timer 0. Timer 0 generates a
//synchronous interrupt once
// every 100Hz or 10 ms.

void timer0_init (void)
{
      EA = 0;
            // disables all interrupts
      timer0_tick = 0;

      TR0 = 0;
            // stops Timer 0

      CKCON &= ~0x08;
            // Timer 0 using CLK24/12 is bit 3=0
      //CKCON = 0x03;
            // this used to be in here - it makes a stretch to 7

            //of strobes (programmable wait states....not sure why

            //it was here...in the original example
```

```
    TMOD &= ~0x0F;
          // clear Timer 0 mode bits
    TMOD |= 0x01;
          // setup Timer 0 as a 16-bit timer

    TL0 = (TIMER0_COUNT & 0x00FF);
          // loads the timer counts
    TH0 = (TIMER0_COUNT >> 8);

    //PT0 = 0;
          // sets the Timer 0 interrupt to low priority
    ET0 = 1;
          // enables Timer 0 interrupt
    TR0 = 1;
          // starts Timer 0
    EA =  1;
          // enables all interrupts
}
```

```
// LISTING E.4.  ezmouse.c

//-----------------------------------------------------------------------
//
//   Main firmware for the APD-SA USB EZUSB chip.  This firmware
//   is downloaded to the EZUSB upon enumeration first as a
//   Cypress UEZUSB device.
//
//   The framework for the necessary functions, initialization codes
//   etc. comes from ezmouse.c and periph.c code from Cypress, among
//   other references. The corresponding information is at the bottom
//   of these comments.  However, extensive coding is specific to
//   the APD-SA USB EZUSB.  This specificity includes:
//
//   - endpoint interrupt functions
//   - USB-DSP communication functions for grabbing and transmitting
//     data
//   - specific initialization options
//
//   Indented comments generally follow the lines to which they refer.
//
//   Some rather large comment sections were included because the
//   browse function in the Keil IDE was not particularly accurate or
//   easily accessible - thus scrolling was often the required method
//   for locating a section of code.
//
//
//
//   Debug information was sent over serial port.  However, most of
//   this code has been commented out because most of the firmware
//   was written, there was little memory left for debug data storage.
//   For this reason, the full Cypress evaluation platform with
//   monitor program would be useful.
//
//   N. Silverman 2004 for MS Thesis
//
//-----------------------------------------------------------------------
//   Original sample framework:
//   Contents:  Hooks required to implement USB peripheral function.
//
//   Copyright (c) 2001 Cypress Semiconductor, Inc.
//   All rights reserved
//-----------------------------------------------------------------------

#pragma NOIV    // Do not generate interrupt vectors

#include <ezusb.h>
#include <ezregs.h>

//#include <stdio.h>
/* prototype declarations for I/O functions */

//mine - APD-SA USB header
#include <apdusb.h>

#define    min(a,b) (((a)<(b))?(a):(b))

#define GD_HID             0x21
#define GD_REPORT          0x22
```

```
#define CR_SET_REPORT      0x09
#define HID_OUTPUT_REPORT 2

#define BTN_ADDR                    0x41

//** my macros, mkay - now included in header file
//#define USB_CONNECT_TO_DSP        0x01
//#define USB_UNRECOGNIZED_COMMAND  0xff
//#define USB_DSP_NO_ACK            0xfe
//#define USB_DSP_ACK 0xbb
//#define USB_REQUEST_VARIABLES     0x02
//#define USB_END_OF_VARIABLE_TRANSFER 0xfd
//#define USB_HELLO                 0xb0

extern BOOLGotSUD;             // Received setup data flag
extern BOOL Sleep;

WORD pHIDDscr;
WORD pReportDscr;
WORD pReportDscrEnd;
extern code HIDDscr;
extern code ReportDscr;
extern code ReportDscrEnd;

void TD_Poll(void);

//Timer functions!!!!
void timer0_init (void);
unsigned timer0_count (void);
void timer0_delay (unsigned count);
void HitWatchdog(void);
//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!___________-----
//========++++++++++++++++++++++@@@@@@@@@@@@@@@@@


BYTE Configuration;        // Current configuration
BYTE AlternateSetting;     // Alternate settings

//**mine, all mine!!!! - some moreAPD-SA USB global variables
BYTE gotfreshdata;
BYTE outcount;
BYTE incount;
BYTE irqcalled;
BYTE receivedatablock;
BYTE g_status;
BYTE index;
BYTE countbytes;

BYTE read_dsp_ack(void);
BYTE send_to_dsp(BYTE val);
//BYTE stop_dsp(void);

//void Dummy(void);
void outsci(char *b);

void outsci(char *b)
{

     int status;
```

```
    SCON0 &= ~0x02; //clear the transmit complete bit

    while (*b != 0 )
    {
          SBUF0 = *b;
          *b++;
          do status = (SCON0); while ((status&0x02)!=0x02);
          //wait until bits are shifted
          SCON0 &= ~0x02; //clear the transmit complete bit
    }

}

/*******************************************************************
**************************************/
/*******************************************************************
**************************************/
/*******************************************************************
**************************************/
/*void Dummy(void)
{
    int i = 0;

    for(i=0; i<0x05; i++)
    {
          i++;
          i--;
    }
}*/ //Dummy()  //Dummy()  //Dummy()  //Dummy()  //Dummy()  //Dummy
()  //Dummy()  //Dummy()  //Dummy()  //Dummy()


//------------------------------------------------------------------
// Task Dispatcher hooks
//   The following hooks are called by the task dispatcher.
//------------------------------------------------------------------
//BOOL enumerated;
/*******************************************************************
**************************************/
/*******************************************************************
**************************************/
/*******************************************************************
**************************************/
//TD_Init()   //TD_Init()   //TD_Init()   //TD_Init()   //TD_Init
()   //TD_Init()   //TD_Init()   //TD_Init()
//TD_Init()   //TD_Init()   //TD_Init()   //TD_Init()   //TD_Init
()   //TD_Init()   //TD_Init()   //TD_Init()
//TD_Init()   //TD_Init()   //TD_Init()   //TD_Init()   //TD_Init
()   //TD_Init()   //TD_Init()   //TD_Init()
void TD_Init(void)                  // Called once at startup
{
    int i=0x00;

    EA=0; //disable interrupts

    //Setup serial port for debug
    PORTCCFG= 0x03;
    //PC0 and 1 = 1 to setup port c pins for peripheral function
```

```
OEC=0xff;//output
//PC0=1;
//PC1=1;
//PCON |= 0x80; //PCON.7 is rate doubler
SCON0 = 0x50;
//SCON0.7 and .6 = 0 and 1 resp for mode 1 on Serial Port 0
//and enable receive
RCAP2L = 0xf3; //overflow for 57600 baud
RCAP2H = 0xff; //overflow for 57600 baud
T2CON = 0x10;
//T2CON.5, .4 and .2 are all 1 for Rx and Tx baud gen and
//start run
T2CON |= 0x04; //start it
CKCON = CKCON |0x20;
//SCON0  = 0x40;//0x50;
/* SCON: mode 1, 8-bit UART, enable rcvr      */
//TMOD |= 0x20;
/* TMOD: timer 1, mode 2, 8-bit reload        */
//PCON |= 0x80; //SMOD0 = 1
//TMOD &= ~0x40; //TMOD6=0
//CKCON |= 0x10; //CKCON4=1
//TH1   = //64;221;
/* TH1:  reload value for 1200 baud @ 16MHz   */
//TR1   = 1;
/* TR1:  timer 1 run                          */
//TI    = 1;
/* TI:   set TI to send first char of UART    */
//*/
//outsci("Init ");
//printf("Hello World\n");

//**This is taken from ep_pair example: two endpoints;
//OUT endpoint is option for HID
//8051 operates only on the even endpoints
// Enable endpoint 2 in, and endpoint 2 out
IN07VAL = bmEP2;          // Validate all EP's
OUT07VAL = bmEP2;

//** From example: int_4pf or something like this:
//** Enable the EP interrupts...?
OUT07IEN |= bmEP2;      // enable ISR for EP2OUT
IN07IEN |= bmEP2;         // and EP2 IN

//** Enable double buffering on endpoint 2 in, and endpoint 2 out
//USBPAIR = 0x09;
//this pairs Bit0: 2IN and 3IN and also (Bit3) pairs 2OUT
//and 3OUT
USBPAIR = 0x01;
//this pairs IN2 and IN3 using IN2 for working location

//** Arm Endpoint 2 out to recieve data - this should
//happen anyway by default on reset
EPIO[OUT2BUF_ID].bytes = 0;

//** NS Added ....? does it work? no. ARM IN EP2
//EPIO[IN2BUF_ID].bytes = 0; don't do this for single
//endpoint, as it will keep the IN busy until IN is received
//and thus will prevent loopback test
```

```
// Setup breakpoint to trigger on TD_Poll()
BPADDR = (WORD)TD_Poll;//(WORD)Dummy; //(WORD)TD_Poll;
//ISR_Ep2in;
USBBAV |= bmBPEN;                    // Enable the breakpoint
USBBAV &= ~bmBPPULSE;     //|= bmBPPULSE;               //

//**From extr_intr.c
PORTACFG = 0x00;
// PORTA is is configured as an I/O
//OEA  = 0xFF;
// PORTA is an output - don't initialize this way -
// it'll bog down the ADC outs if they're running when this inits
OEA =0x00;                  //init as input...
OUTA = 0x00;
// Initialize PORTA to all LOW if used as an output
//PORTCCFG = 0x0C;            // PC2 and PC3 are INT0 and INT1
//PORTBCFG = 0x70;
// PB4, PB5, PB6 are INT4, INT5, INT6
//TCON |= 0x05;
// Detect INT0 and INT1 on fallinfg edge
//OEC |= 0xF3;
//Enable  Interrupts

EIE = 0x01;               // Enable USB Int
//USB_EN_INT6;
// Enable INT6 - do you really want to do this early on?
//probably not.
USB_DIS_INT6;
IE  &= ~0x05;
// Disable External Interrupts 0 and 1

//**From me
PORTBCFG = 0x40;
// PB: 6 is INT6 from 803 PE5 and 7 is still regular IO
// used to signal back to 803 from USB to PE6
// PB5 is to IRQ the 803 (IRQA) to signal connection/
//request etc.
// INT6 is low-to-high transition detection
OEB = 0xa0;
// initialize PB7 and PB5 as outs
OUTB = 0x20;
// IRQ is active low, so initialize the 803 IRQA = EZ PB5 as high
// EICON.3 is the 1=transition detected for INT6
EIP = 0x10;
// EIP.4   =SFR 0xf8 = set to 1 for high priority
// EIP.0 = set to 0 for USB low priority
IP &= ~0x02;
//Timer 0 interrupt set to low priority - watchdog timer
//when set to high priority, it interferes with INT6
//because it has higher natural priority


gotfreshdata = 0x00;
//IN2BUF[0]=0x02;
//IN2BUF[1]=0x03;
//EPIO[IN2BUF_ID].bytes=2;
//EPIO[IN2BUF_ID].cntrl &= ~bmEPBUSY; //clear the busy bit
//IN2BUF[1] = 0xcc; //OUT2BUF[1];
//connect_to_dsp();            //Connect to 803 DSP
```

```
	incount=0;
	outcount=0;
	irqcalled =0;
	receivedatablock=0;
	//not yet armed to receive a data block from dsp via portA
	countbytes=0;
	//OUTB |= 0x80;  //this works out ok

	//for( i=0; i<USB_MAX_BYTES; i++ ){IN2BUF[i] = 0x00;} //clear
IN2EP buffer

	EA  = 1;                      // Enable Global Interrupt

	//this inits a timer currently set to about 100Hz operation
	timer0_init();
} //TD_Init()   //TD_Init()   //TD_Init()   //TD_Init()   //TD_Init()
//TD_Init()   //TD_Init()   //TD_Init()



/*************************************************************************
**************************************/
/*************************************************************************
**************************************/
/*************************************************************************
**************************************/
/*************************************************************************
**************************************/
/*************************************************************************
**************************************/
//read_dsp_ack()   //read_dsp_ack()   //read_dsp_ack()   //read_dsp_ack
()   //read_dsp_ack()   //read_dsp_ack()
//read_dsp_ack()   //read_dsp_ack()   //read_dsp_ack()   //read_dsp_ack
()   //read_dsp_ack()   //read_dsp_ack()
//read_dsp_ack()   //read_dsp_ack()   //read_dsp_ack()   //read_dsp_ack
()   //read_dsp_ack()   //read_dsp_ack()
BYTE read_dsp_ack(void)
{
	BYTE status;
	BYTE count;
	BYTE count2;
	BYTE count3;
	//this connects to the dsp using signalling:
	//803IRQA is EZ PB5 = 0x20; it's active low

	//**This routine should be interrupted by INT6 from the
	//DSP's PEx and have the PINSA set to input
	//and read...they're re-read here anyway...

	//wait for ACK of this IRQ from 803
	count = 0x00;
	count2 = 0x00;
	count3 = 0x00;
	g_status = 0x00;

	outsci("rda ");

	//debug - read the pending interrupts
```

```
IN2BUF[USB_MAX_BYTES-8]=EXIF;
//interrupt flags for INTs 5, 4, I2C, USB (7,6,5,4
//bits respectively)
IN2BUF[USB_MAX_BYTES-9]=EICON;
//bit 3 means INT6 detected low-to-high transition...
//latched ya know.


//USB_EN_INT6 used to be right here ...


//but disable other EP INTs
USB_DIS_USBINT;

//for debug purposes, write this to show where we are...
IN2BUF[USB_MAX_BYTES-2]=0xcc;


//drop 0x20 to signal active low 803 IRQA to request
//INT6 basically if dsp is there and ready
//OUTB &= ~0x20;

//Moving USB_EN_INT6 here allowed proper USB_STOP recognition
//and still seems to allow the
//proper variable transfer too, now deal with problem of no
//STOP ACK after streaming data...

//if we're here, we're waiting for an ACK byte and thus an
//INT6 so enable it at least...
//clear prior INT6
EICON &= ~0x08; //|= 0x08;
//this one is not reset by writing to it...annoying.  clear INT6
//enable INT6
USB_EN_INT6; //this should then be disabled in the INT6 handler
//DANGER here: any pending interrupt will be serviced, even
//if it occured prior and is irrelevent
//to our needs...deal with this ok.
//try this here instead of above to see if it all happens
//too fast before and the response was missed
OUTB &= ~0x20;



do //loop to wait for INT6 and read of ACK byte or time out....
{
      count++;
      if(count>0xfe)
      {
            count=0;
            count2++;
            /*if(count3>0xfe)
            {
                  count2=0;
                  count3++;
                  HitWatchdog(); //maybe
            }*/
      }
      if(count2>0xfe)break;  //or count3...
      if(g_status==USB_DSP_ACK)break;
```

```
	}while(1);
	//await ack byte on portA and watch for too long a time

	//right now, in wait for ACK, in INT6 handler, USB and INT6
	//ints are disabled

	//reset the IRQA803 - it's edge sensitive setting hopefully
	//on the 803 (should be anyway)
	OUTB |= 0x20;

	//for debug purposes...
	IN2BUF[USB_MAX_BYTES-4]=count;
	//used to be count2 but shows fine res
	IN2BUF[USB_MAX_BYTES-5]=count2;
	IN2BUF[USB_MAX_BYTES-6]=g_status;

	if(count2>=0xfe) //it took too long
	{
		IN2BUF[0]=USB_DSP_NO_ACK;
		//re-enable EP INTS
		//USB_EN_USBINT;
		//EPIO[IN2BUF_ID].bytes=64;
		//OUTB |= 0x20; //reset the IRQ pin
		//no response
		//send to USB buffer: no connection to DSP\
		return 0;
	}else{ //ack was returned in time via INT6 so send the
		//ack to the IN2 EP and arm it...then clear the
		//IRQA for the 803
		//OUTB |= 0x80; //tell DSP I heard that ACK
		IN2BUF[0]=USB_DSP_ACK;
		IN2BUF[USB_MAX_BYTES-7]=0xcc;
		//re-enable USB INTs
		//USB_EN_USBINT;
		//EPIO[IN2BUF_ID].bytes=64; //arm IN2EP
		//OK
		//send to USB buffer: connected to DSP
		//OUTB |= 0x20; //reset the IRQ pin
		return 1;
	}

} //read_dsp_ack()   //read_dsp_ack()   //read_dsp_ack()   //
read_dsp_ack()   //read_dsp_ack()   //read_dsp_ack()


/*************************************************************************
**************************************/
/*************************************************************************
**************************************/
/*************************************************************************
**************************************/
BYTE send_to_dsp(BYTE val)
{

	OEA = 0xff;  //make PORTA output - yes, yes, just make it an out
here when a byte needs to be sent.

	OUTA=val;  //set pins to val
```

```
    return 1;   //return
}


/*******************************************************************
**************************************/
/*******************************************************************
**************************************/
/*******************************************************************
**************************************/
//STOP_DSP     //STOP_DSP     //STOP_DSP     //STOP_DSP     //STOP_DSP
//STOP_DSP     //STOP_DSP     //STOP_DSP
//STOP_DSP     //STOP_DSP     //STOP_DSP     //STOP_DSP     //STOP_DSP
//STOP_DSP     //STOP_DSP     //STOP_DSP
//STOP_DSP     //STOP_DSP     //STOP_DSP     //STOP_DSP     //STOP_DSP
//STOP_DSP     //STOP_DSP     //STOP_DSP
/*BYTE stop_dsp(void)
{

    //send a stop to the DSP

    //unsigned int i = 0x0000; //16 bits by compiler docs. -
    //this may cause problems????

    //first disable all interrupts and clear them so as not to
    //fire in the middle of this
    EA=0; //beware this will stop the watchdog interrupt as well

    //let watchdog on dsp timeout
    //for(i=0; i<0xfff; i++)
    //{
    //    HitWatchdog();
    //}

    //need INTs here to read ACK or NACK etc.
    //disable EP ints
    USB_DIS_USBINT;
    //INT6 will be enabled in the read_dsp_ack routine...
    USB_DIS_INT6;

    //clear pending interrupts - I'll need to check if this
    //is necessary in each case...
    IN07IRQ |= 0x04;
    //clear USB EP2IN  int - cleared by writing a 1 to it
    OUT07IRQ |= 0x04;
    //clear USB EP2OUT int -  cleared by writing a 1 to it
    //USBIRQ=0x00;
    EXIF &= ~0x10;//0x00; clear bit 4 = IRQ for USB = USBINT
    EICON &= ~0x08; //|= 0x08;
    //this one is not reset by writing to it...annoying. clear INT6

    //NOTE: calling the read_dsp_ack from this routine which
    //is called by polling routine may cause problems
    //(i.e. linker error: L15) meaning that it's called both
    //from interrupt level and non-interrupt levels
    //meaning potential corruption, however, if it's all setup
    //ok, then it won't be called recursively
    //because interrupts are disabled and cleared within it, and
    //it can't be called until the poll routine
```

```
//is run, which occurs outside of an interrupt routine...
//keep fingers crossed here - the linker doesn't
//realize that the routine itself prevent recursive calling
//from any interrupts so it  warns.  Thanks.

//send an IRQ to DSP, wait for ACK, then
send_to_dsp(USB_STOP);

//signal that this is now command mode, not continuous data
//block mode
receivedatablock = 0;
//this needs to happen prior to read_dsp_ack so that proper
//case is selected
//in that function
//thus, after exit of read_dsp_ack, INT6 should be disabled,
//as well as USB EP ints.

EA=1;
read_dsp_ack(); //==1)      //if we've received the ACK
//{
//    receivedatablock=0; //get ready to receive data block...
//    //USB_DIS_INT6;
//disable INT6 already disabled in the INT6 routine...
//}else{ //no dsp ack
//    receivedatablock=0; //check this....?????
//}
USB_DIS_INT6; //disable any INT6s here...until commands
//require streams of data - just to make sure...


//do it again just to try...
//send an IRQ to DSP, wait for ACK, then
send_to_dsp(USB_STOP);
read_dsp_ack();
//if(read_dsp_ack()==1)      //if we've received the ACK
//{
//    receivedatablock=0; //get ready to receive data block...
//    //USB_DIS_INT6; //disable INT6 already disabled in
//the INT6 routine...
//}else{ //no dsp ack
//    receivedatablock=0; //check this....?????
//}

//EA=0; //disable again just in case...

USB_DIS_INT6; //disable any INT6s here...until
//commands require streams of data - just to make sure...

IN2BUF[1]=USB_STOP; //write this to acknowledge that
//this command was received..

//for debug
IN2BUF[USB_MAX_BYTES-10]=0xcc;

USB_EN_USBINT; //enable future USB interrupts for commands
//from host and transfer of ACK or NACK to host
```

```
    EPIO[OUT2BUF_ID].bytes=0; //may need to place this in the
    //IN2 INT routine to prevent early firing
    EPIO[IN2BUF_ID].bytes=64; //arm IN2EP to send the ACK or
    //NACK bytes
    //EP should be re-armed only at end of USB REQ VARS all
    //acquired so as not have interrupts in the middle

    outsci("eos ");

    //re-enable necessary interrupts
    //EA=1;

} //stop_dsp()  //stop_dsp()//stop_dsp()//stop_dsp()//stop_dsp()//
stop_dsp()//stop_dsp()//stop_dsp()//stop_dsp()
*/

/*******************************************************************
*************************************/
/*******************************************************************
*************************************/
/*******************************************************************
*************************************/
//POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL
//POLL  //POLL  //POLL  //POLL  //POLL  //POLL
//POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL
//POLL  //POLL  //POLL  //POLL  //POLL  //POLL
//POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL
//POLL  //POLL  //POLL  //POLL  //POLL  //POLL
void TD_Poll(void)                  // Called repeatedly while the
device is idle
{

    BYTE i=0x00;
    BYTE isbusy=0x00;

    //don't put a while(1) loop in here, it'll lock up the
    //whole thing. :)


    HitWatchdog(); //like it says...

    //Check OUT2EP just in case there was a stop command issued...
    //but put maximum number of loops for checking for
    //busy buffer...so as not to lock-up the whole stuffs
    //try to just sample once on each loop
    //do
    //{
    //try always arming OUT2BUF for receiving commands.

    //EPIO[OUT2BUF_ID].bytes=0;
    //enable receiving a command here - this is not a good idea
        isbusy=(EPIO[OUT2BUF_ID].cntrl & bmEPBUSY);
        if(isbusy==0) //if not busy
        {
            if(OUT2BUF[0]==USB_STOP)
            {
                EA=0; //hopefully this won't f it up.
                outsci("S ");
                //ok here - it doesn't continually call this
```

```
                    //apparently
                    receivedatablock=0; //reset this to 0 so that in
                    //EP 2 IN and any other INT, proper choice is
                    //selected
                    USB_DIS_INT6; //disable further INT6 responses
                    //to force timeout at dsp end while waiting for
                    //byte acceptance
                    OUT2BUF[0]=0x00; //zero this.
                    //stop_dsp();
                    //nope - instead delay causing dsp to timeout
                    //waiting for byte reception
                    for(isbusy=0; isbusy<0xff;isbusy++)
                    {
                         i=0;
                         do {
                         //using #pragma asm / #pragma endasm with
                         //nop; requires enable SRC option to
                         //generate ASM that then needs to be
                         //compiled to object code
                              i++;
                         } while (i<0xff);
                    } //delay here to cause APDFAIL on the dsp
                    IN2BUF[1]=USB_STOP;
                    //write this to acknowledge that this
                    //command was received..
                    //for debug
                    IN2BUF[USB_MAX_BYTES-10]=0xcc;
                    //this is just for debug info to show we
                    //got here
                    EA=1; //re-enable all enabled interrupts
                    USB_EN_USBINT; //enable USB interrupts
                    outsci("s "); //debug to show we got here
                    EPIO[OUT2BUF_ID].bytes=0; //arm out EP
                    EPIO[IN2BUF_ID].bytes=64; //arm IN EP
                    //outsci("s "); //debug to show we got here
               } //if need to stop
          } //if not busy
     //    i++; //advance loop counter
     //} while(i<0x0f);
     //only do this a few times, or else wait until next loop...

     //outsci("T");

     /*if(receivedatablock==USB_END_OF_VARIABLE_TRANSFER)
     {
          receivedatablock=0; //back to receive commands
          EPIO[OUT2BUF_ID].bytes=0;
          EPIO[IN2BUF_ID].bytes=64;

     }
     */
     //try always arming OUT2 EP for receiving commands
     //EPIO[OUT2BUF_ID].bytes=0; //this appears to cause problems

     //on exit poll, clear breakpoint
     USBBAV &= ~0x08; //USBBAV.3 is breakpoint clear bit
} //POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //POLL  //
POLL  //POLL  //POLL  //POLL  //POLL  //POLL
```

```
BOOL TD_Suspend(void)                // Called before the device goes
into suspend mode
{
     // Turn off breakpoint light before entering suspend
     USBBAV |= bmBREAK;         // Clear the breakpoint
     return(TRUE);
}

BOOL TD_Resume(void)           // Called after the device resumes
{
     return(TRUE);
}

//-----------------------------------------------------------------------
// Device Request hooks
//   The following hooks are called by the end point 0 device request
//   parser.
//-----------------------------------------------------------------------

BOOL DR_ClassRequest(void)
{
     return(TRUE);
}


BOOL DR_GetDescriptor(void)
{
     BYTE length,i;

     pHIDDscr = (WORD)&HIDDscr;
     pReportDscr = (WORD)&ReportDscr;
     pReportDscrEnd = (WORD)&ReportDscrEnd;

     switch (SETUPDAT[3])
     {
           case GD_HID:                          //HID Descriptor
                SUDPTRH = MSB(pHIDDscr);
                SUDPTRL = LSB(pHIDDscr);
                return (FALSE);
           case GD_REPORT:                       //Report Descriptor
                length = pReportDscrEnd - pReportDscr;

                while (length)
                {
                     for(i=0; i<min(length,64); i++)
                          *(IN0BUF+i) = *((BYTE xdata *)
pReportDscr+i);

                     //set length and arm Endpoint
                     EZUSB_SET_EP_BYTES(IN0BUF_ID,min(length,64));

                     length -= min(length,64);

                     // Wait for it to go out (Rev C and above)
                     while(EP0CS & 0x04)
                          ;
```

```
                }
                return (FALSE);
            default:
                return(TRUE);
        }
}

BOOL DR_SetConfiguration(void) // Called when a Set Configuration
command is received
{
        Configuration = SETUPDAT[2];
        return(TRUE);                   // Handled by user code
}

BOOL DR_GetConfiguration(void) // Called when a Get Configuration
command is received
{
        IN0BUF[0] = Configuration;
        EZUSB_SET_EP_BYTES(IN0BUF_ID,1);
        return(TRUE);                   // Handled by user code
}

BOOL DR_SetInterface(void)              // Called when a Set Interface
command is received
{
        AlternateSetting = SETUPDAT[2];
        return(TRUE);                   // Handled by user code
}

BOOL DR_GetInterface(void)              // Called when a Set Interface
command is received
{
        IN0BUF[0] = AlternateSetting;
        EZUSB_SET_EP_BYTES(IN0BUF_ID,1);
        return(TRUE);                   // Handled by user code
}

BOOL DR_GetStatus(void)
{
        return(TRUE);
}

BOOL DR_ClearFeature(void)
{
        return(TRUE);
}

BOOL DR_SetFeature(void)
{
        return(TRUE);
}

BOOL DR_VendorCmnd(void)
{
        return(TRUE);
}

//--------------------------------------------------------------------
// USB Interrupt Handlers
```

```
//   The following functions are called by the USB interrupt jump
//   table.
//-----------------------------------------------------------------------

// Setup Data Available Interrupt Handler
void ISR_Sudav(void) interrupt 0
{
     GotSUD = TRUE;                    // Set flag
     EZUSB_IRQ_CLEAR();
     USBIRQ = bmSUDAV;                 // Clear SUDAV IRQ
}

// Setup Token Interrupt Handler
void ISR_Sutok(void) interrupt 0
{
     EZUSB_IRQ_CLEAR();
     USBIRQ = bmSUTOK;                 // Clear SUTOK IRQ
}

void ISR_Sof(void) interrupt 0
{
     EZUSB_IRQ_CLEAR();
     USBIRQ = bmSOF;                   // Clear SOF IRQ
}

void ISR_Ures(void) interrupt 0
{
     EZUSB_IRQ_CLEAR();
     USBIRQ = bmURES;           // Clear URES IRQ
}

void ISR_IBN(void) interrupt 0
{
   // ISR for the IN Bulk NAK (IBN) interrupt.
}

void ISR_Susp(void) interrupt 0
{
     Sleep = TRUE;
     EZUSB_IRQ_CLEAR();
     USBIRQ = bmSUSP;
}

void ISR_Ep0in(void) interrupt 0
{
}

void ISR_Ep0out(void) interrupt 0
{
}

void ISR_Ep1in(void) interrupt 0
{
}

void ISR_Ep1out(void) interrupt 0
{
}
```

```
//**********************************************************************
****************************************
//**********************************************************************
****************************************
//**|----   |^^^\   /^^^\    ^|^  |\  |
//**|__     |___/      _-^     |   | \ |
//**|____   |       /____    _|_  |  \|
//**********************************************************************
****************************************
//**********************************************************************
****************************************
void ISR_Ep2in(void) interrupt 0
{

     //outsci("EI ");
     //Clear ints
     EZUSB_IRQ_CLEAR();
     IN07IRQ |= 0x04;
           //clear the IRQ for EP2 IN   although it looks like setting
           //a bit, it clears the IRQ


     //if this request has been hit, a packet from IN2BUF has
     //been transmitted, thus
     //EP2 IN data to host is now old data
     //this interrupt signals that one buffer is free
     gotfreshdata = 0x00;
     //IN2BUF[0]=IN2BUF[0]+1;
     //EPIO[IN2BUF_ID].bytes=2;
     //this also should mean that the busy bit is now cleared
     //until the byte count is set again
     //which will set the busy bit
     //buffer now empty
     incount++;
     //EPIO[IN2BUF_ID].bytes=2; //re-arm ???
     //EA=1; //enable interrupt only after ACK picked up just prior
     //to variable transfer - impossible...you'll never
     //get here without an interrupt...so just disable
     //INT4 interrupt...prior, then re-enable here
     if(receivedatablock==1)
     {
           outsci("EI1 ");
           //re-enable INT6 during data transfer to allow for
           //next transfer
           USB_EN_INT6;

           //if we've finished total transfer reset the
           //receievdata block, thus
           //INT6 won't be re-enabled until another command set
           //is received...
           if(g_status == USB_END_OF_VARIABLE_TRANSFER)
                receivedatablock=0;
     }

     if(receivedatablock==2)
     {
           outsci("EI2 ");
           USB_EN_INT6;
     }
```

```
}


//*********************************************************************
***************************************
//*********************************************************************
***************************************
//**|----   |^^^\  /^^^\    /^^^\ |   |  ^^^|^^^
//**|__     |___/    _-^    |   | |   |     |
//**|____   |      /____    \___/ \___/     |
//*********************************************************************
**************************************
//*********************************************************************
**************************************
void ISR_Ep2out(void) interrupt 0
{


      BYTE i=0x00;
      //this interrupt signals that data was received from host
      //thus we have fresh data
      //outsci("E2O ");
      gotfreshdata = 0x01;
      outcount++;
      //bmBusy should be 0 here...cleared on successful receipt of data
      //writing the byte count reg should arm endpoint and cause
      //busy bit to go

      IN2BUF[USB_MAX_BYTES-1] = outcount;

      switch (OUT2BUF[0])
      //report ID is not trasmitted on bus because only one
      //report...thus byte 0 is first data byte
      {

            case USB_CONNECT_TO_DSP:
                  receivedatablock=0; //added 10.8.03
                  for( i=0; i<USB_MAX_BYTES; i++ ){IN2BUF[i] = 0x00;}
                  //clear IN2EP buffer
                  send_to_dsp(USB_HELLO);
                  i=read_dsp_ack();
                  USB_EN_USBINT;
                  EPIO[IN2BUF_ID].bytes=64; //arm IN2EP
                  EPIO[OUT2BUF_ID].bytes = 0;
                  //arm the EP to receive more data
                  //EA=1; //re-enable interrupts - disables after ACK
                  //read; if no ACK, then never disabled
                  //right now INT6 should be disabled, but USB INTS
                  //enabled and IN2EP armed
                  //host program should not send OUT until the IN has
                  //been received, so another OUT2INT shouldn't be
                  //generated
                  //prior to the IN2EP INT which will then re-enable
                  //INT6
                  //unless, host automatically fires an OUT
                  //often....we'll see
                  //???? if INT6 fires first (erroneously, without any
                  //other pending command, what happens?
```

```
        outsci("UC ");
        break; //return;

    case USB_REQUEST_VARIABLES:
        //USBBAV |= bmBPEN;
        // clear the breakpoint
        receivedatablock=0;
        for( i=0; i<USB_MAX_BYTES; i++ ){IN2BUF[i] = 0x00;}
        //clear IN2EP buffer
        send_to_dsp(USB_REQUEST_VARIABLES);
        i=read_dsp_ack();
        if(i==1)      //if we've received the ACK
        {
            receivedatablock=1;
            //get ready to receive data block...
            //USB_DIS_INT6; //disable INT6 already disabled
            //in the INT6 routine...
        }else{ //no dsp ack
        }
        USB_EN_USBINT;
        EPIO[IN2BUF_ID].bytes=64; //arm IN2EP
        EPIO[OUT2BUF_ID].bytes=0;
        //may need to place this in the IN2 INT routine to
        //prevent early firing
        //EA=1; //re-enable interrupts - disables after ACK
        //read; if no ACK, then never disabled; allow EPINTs
        //EP should be re-armed only at end of USB REQ VARS
        //all acquired so as not have interrupts in the middle
        outsci("UR ");
        break;

    case USB_STREAM_ADC:
        for( i=0; i<USB_MAX_BYTES; i++ ){IN2BUF[i] = 0x00;}
        //clear IN2EP buffer
        send_to_dsp(USB_STREAM_ADC);
        if(read_dsp_ack()==1)
        //if we've received the ACK
        {
            receivedatablock=1;
            //get ready to receive data block...
            //USB_DIS_INT6;
            //disable INT6 already disabled in the INT6
            //routine...
        }else{ //no dsp ack
        }
        USB_EN_USBINT;
        EPIO[IN2BUF_ID].bytes=64;
        //arm IN2EP to send the ACK or NACK bytes
        EPIO[OUT2BUF_ID].bytes=0;
        //may need to place this in the IN2 INT routine
        //to prevent early firing
        //EA=1; //re-enable interrupts - disables after ACK
        //read; if no ACK, then never disabled; allow EPINTs
        //EP should be re-armed only at end of USB REQ VARS
        //all acquired so as not have interrupts in the middle
        outsci("US ");
        break;

    case USB_GET_HOST_VARS:
```

```
        USBBAV |= bmBPEN;
        // clear the breakpoint
        for( i=0; i<USB_MAX_BYTES; i++ ){IN2BUF[i] = 0x00;}
        //clear IN2EP buffer
        send_to_dsp(USB_GET_HOST_VARS);
        if(read_dsp_ack()==1)
        //if we've received the ACK
        {
             receivedatablock=2;
             //get ready for sending variables on INT6
             //USB_DIS_INT6; //disable INT6 already disabled
             //in the INT6 routine...
        }else{ //no dsp ack
        }
        USB_EN_USBINT;
        EPIO[IN2BUF_ID].bytes=64; //arm IN2EP
        EPIO[OUT2BUF_ID].bytes=0;
        //may need to place this in the IN2 INT routine to
        //prevent early firing
        //EA=1; //re-enable interrupts - disables after ACK
        //read; if no ACK, then never disabled; allow EPINTs
        //EP should be re-armed only at end of USB REQ VARS
        //all acquired so as not have interrupts in the middle
        outsci("UG ");
        break;

   case USB_RUN:
        for( i=0; i<USB_MAX_BYTES; i++ ){IN2BUF[i] = 0x00;}
        //clear IN2EP buffer
        send_to_dsp(USB_RUN);
        if(read_dsp_ack()==1)
        //if we've received the ACK
        {
             receivedatablock=1;
             //get ready to receive data block...
             //USB_DIS_INT6;
             //disable INT6 already disabled in the INT6
             //routine...
        }else{ //no dsp ack
        }
        USB_EN_USBINT;
        EPIO[IN2BUF_ID].bytes=64;
        //arm IN2EP to send the ACK or NACK bytes
        EPIO[OUT2BUF_ID].bytes=0;
        //may need to place this in the IN2 INT routine to
        //prevent early firing
        //EA=1; //re-enable interrupts - disables after ACK
        //read; if no ACK, then never disabled; allow EPINTs
        //EP should be re-armed only at end of USB REQ VARS
        //all acquired so as not have interrupts in the middle
        outsci("fRUN ");
        break;

   //OK, but what if INTs are enabled, e.g. CONNECT works,
   //but not STOP...

   default:
        //may need to check if IN2 is busy, but we'll assume
        //things are queued properly
```

```
                //if(!(EPIO[IN2BUF_ID].cntrl & bmBusy))
                IN2BUF[0] = USB_UNRECOGNIZED_COMMAND;
                EPIO[OUT2BUF_ID].bytes = 0;
                //arm the EP to receive more data
                //EPIO[OUT2BUF_ID].bytes=0;
                EPIO[IN2BUF_ID].bytes=64; //arm the IN2 EP for data to
                //send; this sets the busy bit as well supposedly
                outsci("UU ");
                break;//return;
     }

     //since out has been read, arm it for another write
     //EPIO[OUT2BUF_ID].bytes=0;

     //Clear ints
     EZUSB_IRQ_CLEAR();
     OUT07IRQ |= 0x04; //clear the IRQ for EP2 OUT
     //although it looks like setting a bit, it clears the IRQ

     //reset the buffer
     //OUT2BUF[0] = 0x00; //maybe??



}

void ISR_Ep3in(void) interrupt 0
{
}

void ISR_Ep3out(void) interrupt 0
{
}

void ISR_Ep4in(void) interrupt 0
{
}

void ISR_Ep4out(void) interrupt 0
{
}

void ISR_Ep5in(void) interrupt 0
{
}

void ISR_Ep5out(void) interrupt 0
{
}

void ISR_Ep6in(void) interrupt 0
{
}

void ISR_Ep6out(void) interrupt 0
{
}

void ISR_Ep7in(void) interrupt 0
```

```
{
}

void ISR_Ep7out(void) interrupt 0
{
}
```

APPENDIX F: APD200 SOFTWARE – REVISED PORTIONS OF APD100

Code Listings:

- Listing F.1. ApdUsb.bas – main APD200 code that provides USB functions

```
' LISTING F.1.  ApdUsb.bas

Attribute VB_Name = "ApdUsb"
Option Explicit

'-----------------------------------------------------------------------
'
' ApdUsb.bas
'
' This code contains most of the functions that support the APD-SA
' USB device as an HID class USB device.  Modifications have been
' made to other modules and forms in the APD PC host code, but the
' majority of functions are for the APD-SA USB are in this module.
'
' This code works as two layers, the first providing a similar
' interface to APD functions as was found in the APD100 model,
' the second layer working below the first to handle USB controls.
'
' There are several routines that provide data manipulation from the
' USB data stream in order to extract variables and use the APD
' functions in the first layer.
'
' The USB access functions are built around the Windows HID device
' driver that expects to communicate with a USB 1.1 compliant USB
' device.
'
' The basic HID-class USB device functions in this code are adapted
' from:
' Project: HIDDemo2.vbp
' Version: 1.1
' Date: 3/8/00
' Copyright 1999 and 2000 by Jan Axelson (jan@lvr.com)
'
' This example USB HID projects includes the functions:
'  DisplayResultofAPICall
'  DoAWriteFile
'  FindtheHID
'  GetDataString
'  GetDeviceCapabilities
'  GetDeviceString
'  ReadSingleReport
'  SendReporttotheHID
'
' The above listed functions communicate with the Windows HID-class
' USB device driver
' and require J. Axelson's ApiDeclarations file.
'
' This API declaration file however
' has been expanded
' to access more functions in the HID driver that are implemented in
' this APD program.
'
' Further the example functions have been expanded to better suit
' the device here.
'
' The vendor ID and device ID have been changed to match that
' programmed in the
' EZ-USB chip
```

```
' The USB Re-numeration code has been adapted from
' examples provided by Cypress (EZ-USB) for the AN2131 chip
' This includes the functions:
'  Download
'  DoneDownload
'  All functions in EZUSB.bas module
'  All functions in Memory module
'
' However, the code has been changed to be implemented specifically
' for the APD-SA device
'
' The remaining functions, and changes to the above examples were
' written to
' enable the APD-SA USB functionality within the existing APD100
' program
'
' Much has been changed in the APD100 program thus making reasonable
' its promotion to
' APD200
'
' The original data acquisition card functions have been left intact
' or modified
' to check for conditions of APD-SA attachment - thus new or modified
' routines have
' been written for all aspects of the APD program including:
'  Data collection during run, calibration, variable transfer, etc.
'  Device Calibration
'  Data display on panel
'  Plotting
'  Saving or logging data to file
'
' The bulk of new code related to USB functionality resides within
' this ApdUsb module
' however much modified code and a few added routines reside within
' the ApdMain
' module, ApdPlotting module or on forms (code section)
' ApdIndic, ApdSim, ApdTesing modules and associated forms were
' designed for
' prior to the development of the final prototype of the APD-SA for
' testing the APD,
' simulating an APD panel, and simulating a fast response APD display.
'
' In general, comments are either line by line of for a section.
' Line comments generally follow the line to which they refer.
'
'  N. Silverman 2004 for MS Thesis
'
'-------------------------------------------------------------------

Dim Capabilities As HIDP_CAPS
Dim DataString As String
Dim DetailData As Long
Dim DetailDataBuffer() As Byte
Dim DeviceAttributes As HIDD_ATTRIBUTES
Dim DeviceDetected As Boolean
Dim DevicePathName As String
Dim DeviceInfoSet As Long
Dim ErrorString As String
Dim HID As Long
```

```
Dim LastDevice As Boolean
Dim MyDeviceDetected As Boolean
Dim MyDeviceInfoData As SP_DEVINFO_DATA
Dim MyDeviceInterfaceDetailData As SP_DEVICE_INTERFACE_DETAIL_DATA
Dim MyDeviceInterfaceData As SP_DEVICE_INTERFACE_DATA
Dim Needed As Long
Dim OutputReportData(7) As Byte
Dim PreparsedData As Long
Dim ReadBuffer() As Byte
Dim ReportNumber As Integer
Dim result As Long
Dim SendBuffer() As Byte
Dim Timeout As Boolean
'Dim WithEvents HIDObj As clsHID
'by niz
Dim CurrentCommTimeout As COMMTIMEOUTS

Public blnKill As Boolean


'For download
Public strBuffer As String

' For overlapped reads...
Public EventObject As Long
Public HIDOverlapped As OVERLAPPED

'For typical SAAPD values:
'#2 Fleisch: 1 mmH20 = 0.346 lps; 10 mmH20 = 3.253 lps
'Flow sensor: +/- 1 inH20 for full range (0.25 to 4.25 volts out)
'Pressure sensor: +/- 10 inH20 for full range (0.25 to 2.25 volts out)
'For zero point for transducers nears: 7400 and lowest reading
'at abour 480 counts:
'Flowspan: -.001186 lps / count
'Pressspan: -.00367 cmH20 / count



'Set these to match the values in the device's firmware and INF file.
Const MyVendorID = &H547
'this is reversed in windows &H4705 '4705 from DSCR.A51 file '&H4242
'&HABF  '&H925
Const MyProductID = &H1028  '&H2810 also reversed bytes
'2810 from DSCR.A51 file '&H4200 '&H3B9 '&H1234
Const USB_CONNECT_TO_DSP = &H1
'constant sent to USB to say "hey buddy, connect to DSP, mkay?
Const USB_UNRECOGNIZED_COMMAND = &HFF
'USB says what are trying to say?
Const USB_DSP_ACK = &HBB
'says yes, I received the ACK byte from the DSP ok?
Const USB_DSP_NO_ACK = &HFE
'dsp did not ack in time upon connect trial
Const USB_REQUEST_VARIABLES = &H2
'Ask for all relevant variables from DSP
Const USB_END_OF_VARIABLE_TRANSFER = &HFD
'last variables received - end of transfer
Public Const USB_BYTES_PER_VAR = 6
'number of bytes composing a variable in the buffer
Const USB_MAX_VARS_PER_BUFFER = 10
```

```
Const USB_NUM_VARS = 36 'total number of variables to transfer etc.
Const USB_MAX_READ_REPORTS = 10
'total number of allowed reports to read
Const USB_MAX_ATTEMPTED_REPORTS = USB_MAX_READ_REPORTS + 20
Const USB_DEC_PT = 46 'decimal point in ASCII
Const USB_NEG_SIGN = 45 'hyphen in ASCII
Const USB_STREAM_ADC = &H3
Const USB_RUN = &H6
'set APD USB to runmode and collect data - this is just pert val, and
'press and flow vals
Const USB_STOP = &HFB 'signal stop to the dsp to stop streaming to usb
Const USB_GET_HOST_VARS = &H5
'request for host computer to send vars to apd for storage e.g.
'calibration
Const USB_MAX_STREAM_SIZE = 5 * 2 * 500
'5 seconds x 500Hz x 2 data channels
Public Const USB_MAX_RUN_SIZE = 1 * 4 * 500
'1 second x 4 data slots x 500Hz
Public Const USB_BYTES_PER_REPORT = 64
'Really there are 65 (0 to 64) with the 0th being the report ID,
'but each
'report only contains 64 actual data bytes (1 to 64)
Const USB_HOST_TOO_SLOW = &HFA
'for when the host is collecting data too slowly in the stream or run
'modes this is signalled from the TimeProc, not the subroutines
'that transfer the data
Public Const USB_RUN_VARS = 3 'three variables transmitted
Const USB_RUN_FULL = 7
'this is for running with full data returned after each pert.
Const USB_ERR_PAD = 0.005
'error padding for transfer of rd for 0.0004 -> 0.0
Const USB_NUM_INPUT_BUFFERS = 640
'64 reports in buffer - circular buffer - default is 8

Public Const USB_DEF_MIN_FREQ = 5 'Hz
Public Const USB_DEF_MAX_FREQ = 14 'Hz

Public Vars() As Single
'array of variables from buffer to be redimmed when needed
Public intVarsIndex As Integer 'index for this vars array
Public Digits() As Integer
Public intPertNo As Integer



'*********************************************************************
'*********************************************************************
Public Sub ToggleApdUsbOptions(blnUsbInUse As Boolean)

' This sub switches available options on menus on ApdPanel
' and dialog boxes, depending on which devices (USB or APD100)
' is in use.

With ApdCal

    If blnUsbInUse = True Then
        .fmeApdUsb.Enabled = True
        .txtDuty.Enabled = True
        .cmdDownloadCal.Enabled = True
```

```
        .cmdRetrieveCal.Enabled = True
        .Frame4.Enabled = False
        .Frame5.Enabled = False
    ElseIf blnUsbInUse = False Then
        .fmeApdUsb.Enabled = False
        .txtDuty.Enabled = False
        .cmdDownloadCal.Enabled = False
        .cmdRetrieveCal.Enabled = False
        .Frame4.Enabled = True
        .Frame5.Enabled = True
    End If

End With

With ApdPanel

    If blnUsbInUse = True Then
        .Simulation.Enabled = False
        .Simulate.Enabled = False
        .Testing.Enabled = False
        .SaveFFTData.Enabled = False
        .SaveRXValues.Enabled = False
        .SaveLast5Sec.Enabled = False
        .FftResTime.Enabled = False
        .ResistReact.Enabled = False
        .RvsVolume.Enabled = False
        .MagnitudePlot.Enabled = False
    ElseIf blnUsbInUse = False Then
        .Simulation.Enabled = True
        .Simulate.Enabled = True
        .Testing.Enabled = True
        .SaveFFTData.Enabled = True
        .SaveRXValues.Enabled = True
        .SaveLast5Sec.Enabled = True
         .FftResTime.Enabled = True
        .ResistReact.Enabled = True
        .RvsVolume.Enabled = True
        .MagnitudePlot.Enabled = True
    End If

End With

End Sub


'*******************************************************************
'*******************************************************************
Public Sub OutDecFloat(floatin As Single, Optional ByRef
intSendBufIndex As Integer = -1)

' This sub sends out a float (single) over USB
'

'This method uses string conversion and dumping of each character
Dim i As Integer
Dim j As Integer
Dim stringval As String
Dim dig As String
Dim blnDidDecPt As Boolean
```

```
ReDim Digits(USB_BYTES_PER_VAR - 1)

blnDidDecPt = False

stringval = CStr(floatin)

If Len(stringval) > 0 Then
For i = 0 To Len(stringval) - 1
'hoping that it's always shorter than the Digits
    If i >= USB_BYTES_PER_VAR - 1 Then Exit For
    dig = Mid(stringval, i + 1, 1)
    If dig = "-" Then
        Digits(i) = USB_NEG_SIGN
    ElseIf dig = "." Then
        Digits(i) = USB_DEC_PT
        blnDidDecPt = True
    ElseIf dig = "" Then
        Digits(i) = 0
    Else
        Digits(i) = CInt(dig)
    End If
Next i

If ((Len(stringval) - 1) < (USB_BYTES_PER_VAR - 1)) _
And blnDidDecPt = False Then
    Digits(i) = USB_DEC_PT
     'Add decimal point after integers that are less than 6 places
End If
'i should be incremented arleady as a result of exiting the for loop

Debug.Print "Floatin: " & CStr(floatin)

For i = 0 To UBound(Digits)
    Debug.Print i & ": " & CStr(Digits(i))
    If intSendBufIndex <> -1 Then
        SendBuffer(intSendBufIndex) = Digits(i)
        intSendBufIndex = intSendBufIndex + 1
    End If
Next i

End If

End Sub


'*******************************************************************
'*******************************************************************
Public Sub RezeroUSB()

' This sub rezeroes the APD-SA USB by collecting five seconds
' worth of data from both ADC channels
' The values are averaged to calculate a zero point (the voltage
' or counts really when no pressure or flow is applied)


    Dim newxo As Single
    Dim msgTxt As String
```

```
    Dim response As Variant

    'Start acquisition of five seconds.
    MsgBox "Apply no flow or pressure to APD for five seconds."

    Beep

    MyMsg ("Reading pressure and flow values...")

    ApdUsb.StreamADC

    MyMsg ("...")

    newxo = Average(mp()) 'Ch1 of 2
    msgTxt = "The mouth pressure baseline changed from " & Format(xp,
"0")
    msgTxt = msgTxt & " to " & Format(newxo, "0")
    msgTxt = msgTxt & " ADC Counts. Update the calibration?"
    response = MsgBox(msgTxt, vbYesNo, "Calibration")
    If (response = vbYes) Then
        xp = CSng(Round(newxo))
        ApdUsb.GetHostVars 'write values to the USB flash
    Else
        MsgBox "Not updated."
    End If

    newxo = Average(fr()) 'Ch2 of 2
    msgTxt = "The flowrate baseline changed from " & Format(xo, "0")
    msgTxt = msgTxt & " to " & Format(newxo, "0")
    msgTxt = msgTxt & " ADC Counts. Update the calibration?"
    response = MsgBox(msgTxt, vbYesNo, "Calibration")
    If (response = vbYes) Then
        xo = CSng(Round(newxo))
        ApdUsb.GetHostVars 'write values to the USB flash
    Else
        MsgBox "Not updated."
    End If

    'This will re-download the variables - can then ensure
    'they were transferred ok if necess.
    ApdUsb.RequestVariables

    'Re-connect just to reset if necess.
    ApdUsb.Connect


End Sub


'******************************************************************
'******************************************************************
Sub CalibratePressureUSB()

' This sub performs the pressure calibration routine for the
' APD-SA USB.  It collects five seconds worth of ADC data from
' both channels and then computes calibration coefficients
' based on average values and the specified actual pressure
' Upon completion and acceptance of the values by the user, the
' calibration values are sent back to the APD-SA to be written to
```

```
' FLASH memory.

    Dim sum As Long, rw As Variant, chn As Integer
    Dim span As Variant, offset As Variant, x As Variant
    Dim X1 As Single, X2 As Single 'A/D calibration readings.
    Dim Y1 As Single, Y2 As Single 'Applied calibration pressures.
    Dim txt As String, msgTxt1 As String, msgTxt2 As String
    Dim userInput As Variant
    Dim response As Variant

    msgTxt1 = "Apply low mouth pressure and press OK (cmH2O):"
    msgTxt2 = "Apply high mouth pressure and press OK (cmH2O):"

    'Ask for low pressure calibration point.
    userInput = InputBox(msgTxt1, "Calibration", "0.00")
    Y1 = Val(userInput)

    MyMsg ("Reading low pressure...")
    'Start acquisition of five seconds.
    ApdUsb.StreamADC

    Beep
    X1 = Average(mp())
    MyMsg ("...")

    'Ask for high pressure calibration point.
    userInput = InputBox(msgTxt2, "Calibration", "10.0")
    Y2 = Val(userInput)

    MyMsg ("Reading high pressure...")
    'Start acquisition of five seconds.
    ApdUsb.StreamADC

    Beep
    X2 = Average(mp())
    ApdPanel.StatusBar.Caption = "..."

    'Calculate calibration values.
    If (Abs(Y1 - Y2) < 0.01 Or Abs(X1 - X2) < 0.01) Then
        MsgBox "Calibration failed."
        Exit Sub
    End If
    span = (Y2 - Y1) / (X2 - X1)
    'offset = (0 - Y1) / ap + X1 'In case y1 is not 0.
    offset = X1 - ((Y1 - 0#) / span)

    'Display results to user
    txt = "Your calibration value was " & Format(span, "0.000000")
    txt = txt & " cmH2O/Count. Update the calibration?"
    response = MsgBox(txt, vbYesNo, "Calibration")

    'If accept the updated cal, then save the values in program and to
    'the USB device
    If (response = vbYes) Then
        xp = CSng(Round(offset, 0))
        ap = CSng(Round(span, 8))
       'this depends on format for the span used....check precision
       'etc.
        ApdUsb.GetHostVars 'write values to the USB flash
```

```
    Else
        MsgBox "Not updated."
    End If


End Sub





'*********************************************************************
'*********************************************************************
Public Sub CalibrateFlowBySyringeUSB()

' This sub performs flow calibration on APD-SA USB by
' collecting five seconds worth of ADC data from both channels.
' The calibration coefficient is calculated by integrating the
' ADC counts over time based upon the specified total injection
' and withdrawal volumes.
' Value are calculated for inhalation and exhalation, but as of this
' writing, the average is taken and applied to both directions.
' Upon completion and acceptance of the values by the user, the
' calibration values are sent back to the APD-SA to be written to
' FLASH memory.

    Dim msgTxt As String
    Dim actVol As Variant
    Dim inVol As Single, exVol As Single
    Dim response As Variant

    'Setup collection of baseline data.  One second average.
    MsgBox "Apply no flow through the APD and press OK."

    MyMsg ("Reading baseline...")

    ApdUsb.StreamADC

    Beep

    xo = Average(fr())
    MyMsg ("...")

    'User injects volume through APD.
    Do
        msgTxt = "What calibration volume will you use? Liters:"
        actVol = InputBox(msgTxt, "Calibration", "3.0")
    Loop Until IsNumeric(actVol) Or actVol = vbCancel
    If actVol = vbCancel Then Exit Sub

    actVol = Val(actVol)

    MsgBox "Press OK and then inject the calibration volume within 5
seconds."

    MyMsg ("Monitoring Flow...")

    ApdUsb.StreamADC
```

```
    Beep
    MyMsg ("...")

    exVol = (Average(fr()) - xo) * 5 'Average flowrate*5 seconds.

    negIsExh = (exVol < 0)

    MsgBox "Press OK and then withdraw the calibration volume within 5
seconds."

    MyMsg ("Monitoring Flow...")

    ApdUsb.StreamADC

    Beep

    MyMsg ("...")

    inVol = (Average(fr()) - xo) * 5 'Average flowrate*5 seconds.

    If (Abs(inVol) > 0.01 And Abs(exVol) > 0.01) Then
        ai = -actVol / inVol
        ae = actVol / exVol
        msgTxt = "Your inhalation calibration was " _
           & Format(ai, "0.000000")
        msgTxt = msgTxt & " Lps/count and your exhalation"  _
           & calibrations was " & Format(ae, "0.000000")
        msgTxt = msgTxt & " Lps/count. Update values?"
        response = MsgBox(msgTxt, vbYesNo)
        'If accept the updated cal, then save the values
           'in program and to the USB device
        If (response = vbYes) Then
            xo = CSng(Round(xo, 0))
            ai = CSng(Round((Abs(ae) + Abs(ai)) / 2#, 8))
           'check for rounding here depending on format
            'If (exVol < 0) Then ai = ai * -1# 'this is unneeded
            ae = ai
            ApdUsb.GetHostVars 'write values to the USB flash
        Else
            MsgBox "Not updated on stand-alone APD."
        End If
    Else
        MsgBox "Calibration error.  No flow detected."
    End If

End Sub


'*********************************************************************
'*********************************************************************
Public Sub OutDecFloatMath(floatin As Single)

' This sub sends out a float (single) to the APD-SA USB
' over USB by math (rather than a text approach).  Compare
' with program OutDecFloat.

ReDim Digits(USB_BYTES_PER_VAR)

'Also note that when viewing locals and when program runs for that
```

```
'matter, dimming all ints
'in one line does not create equivalent types..... same with singles
'etc.  so dim each
'separately here to be sure

Dim rounddivisor As Single
Dim multcounter As Single
Dim copyfloat As Single
Dim keepmult As Single
Dim floatval As Single
Dim digit As Integer
Dim digitplace As Integer
Dim num10divs As Integer
Dim totdigout As Integer
Dim prevaccum As Long
Dim value As Long
Dim i As Integer
Dim digindex As Integer

'This routine appears to work most of the time and all the time in '
'theory, but
'cannot be used in this program (use alternative method) because VB
'doesn't
'do 'Fix' properly -> e.g. 0.53 * 100 will fix to 52 and then screw up
'the whole method.

rounddivisor = 0.5
multcounter = 10#
copyfloat = floatin

digit = 0
digitplace = 0 'USB_BYTES_PER_VAR - 1
num10divs = 0
totdigout = USB_BYTES_PER_VAR
prevaccum = 0
value = 0
i = 0
digindex = 0

If floatin < 0 Then
    digit = USB_NEG_SIGN
    Digits(digindex) = digit
    digindex = digindex + 1
    floatin = floatin * -1#
End If

keepmult = 10#

'First count number power of ten
Do While (Fix(floatin / multcounter) <> 0)
    num10divs = num10divs + 1
    multcounter = multcounter * 10#
    keepmult = multcounter
Loop

'Loop to output each digit
For i = 0 To num10divs
    rounddivisor = rounddivisor / 10#
Next i
```

```
floatin = floatin / multcounter
'floatin = floatin + rounddivisor
multcounter = 10#

Do
    prevaccum = prevaccum * 10#
    floatval = floatin * multcounter 'Fix doesn't really work properly
    value = Fix(floatval)
    digit = Fix(value - prevaccum)
    If multcounter = 10# * keepmult Then
        Digits(digindex) = USB_DEC_PT
        digindex = digindex + 1
    End If
    multcounter = multcounter * 10
    prevaccum = value

    'Just in case added decimal point in last place of array
    If digindex >= USB_BYTES_PER_VAR Then Exit Do

    digitplace = totdigout - i
    'output decimal point if necessary

    Digits(digindex) = digit
    digindex = digindex + 1
Loop Until digindex >= USB_BYTES_PER_VAR

End Sub


'*******************************************************************
'*******************************************************************
Private Sub ShowReadBuffer()

' This sub is primarily for debugging and prints
' to the APDUSB monitoring panel (which is not used
' in the release version of APD200) the values of the
' USB buffer that was read most recently.

Dim intCount As Integer
Dim strLine As String

With ApdUSBPanel.lstUSB

.AddItem "ReadBuffer contents:"
.ListIndex = .ListCount - 1

For intCount = 0 To UBound(ReadBuffer)
    strLine = strLine & Hex$(intCount) & ": " & Hex$(ReadBuffer
(intCount)) & Chr(9)   'this is tab
    If ((intCount + 1) Mod 16 = 0) Then
     'if its a multiple of 16 thensesx
        'strLine = strLine & Chr(10) & Chr(13)
           'linefeed and carriage return ASCII
        .AddItem strLine
        .ListIndex = .ListCount - 1
        strLine = ""
    End If
```

```
Next intCount

'Add last line if not multiple of 16
.AddItem strLine
.ListIndex = .ListCount - 1


End With


End Sub



'*******************************************************************
'*******************************************************************
Private Sub ShowSendBuffer()

' This sub is primarily for debugging and prints
' to the APDUSB monitoring panel (which is not used
' in the release version of APD200) the values of the
' USB buffer that was sent most recently.

Dim intCount As Integer
Dim strLine As String

With ApdUSBPanel.lstUSB

.AddItem "SendBuffer contents:"
.ListIndex = .ListCount - 1



For intCount = 0 To UBound(SendBuffer)
    strLine = strLine & Hex$(intCount) & ": " & Hex$(SendBuffer
(intCount)) & Chr(9)   'this is tab
    If ((intCount + 1) Mod 16 = 0) Then
     'if its a multiple of 16 thensesx
        'strLine = strLine & Chr(10) & Chr(13)
           'linefeed and carriage return ASCII
        .AddItem strLine
        .ListIndex = .ListCount - 1
        strLine = ""
    End If
Next intCount


End With


End Sub


'*******************************************************************
'*******************************************************************
Public Sub StopDsp()

' This sub sends a stop command via USB to the DSP
' The stop command can be for a stop sending ADC data
```

```
' or stop running

    Dim blnResult As Boolean
    '
    '
    'Assume the WriteFile function will set SendBuffer(0) =
    'ReportID to zero
    SendBuffer(1) = USB_STOP

    'Clear input report queue - Gets rid of the reports waiting in
     'ring buffer
    'that have not yet been read...
    blnResult = HidD_FlushQueue(HID)
    ApdUSBPanel.lstUSB.AddItem "Flush Queue Result: " & CStr(blnResult)
    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
    '
    '
    blnResult = DoAWriteFile()
    '
    'Try the following here: it may flush the response though, in
     'which case, maybe put flush
    'earlier and try reading several reports in the queue.
    'However, may be slow enough (delayed enough) response that queue
     'is flushed and then response
    'can be read. - !!!!!!!!!!!!NOTE is there a way to stop reading
     'reports? (i.e. tell
    'the driver to stop :)


    If blnResult = True Then
     'wrote file OK, so check for DSP ACK connection etc.
        ApdUSBPanel.lstUSB.AddItem "Wrote USB_STOP OK. Waiting for
acknowledgement from DSP."
        '
        'Clear input report queue - Gets rid of the reports waiting in
           'ring buffer
        'that have not yet been read...
        'blnResult = HidD_FlushQueue(HID)
        'ApdUSBPanel.lstUSB.AddItem "Flush Queue Result: " & Cstr
           '(blnResult)
      'ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
        '
        'Do ReadFile waiting for ACK
        blnResult = (ReadSingleReport() > 0)
           'ReadSingleReport returns the number of bytes read
        '
        'If...then write ok...connected to USB APD
        If blnResult = True Then
            ApdUSBPanel.lstUSB.AddItem "Read something here..."
            ApdUSBPanel.lstUSB.AddItem "Result: " & Hex$(ReadBuffer(1))
& ", outcount: " & Hex$(ReadBuffer(2))
           'assuming (0) is report ID ?
            If ReadBuffer(1) = USB_DSP_NO_ACK Then
                ApdUSBPanel.lstUSB.AddItem "= USB_DSP_NO_ACK..."
            End If
            If ReadBuffer(1) = USB_DSP_ACK Then
                ApdUSBPanel.lstUSB.AddItem "DSP ACK'd USB_STOP."
            End If
            If ReadBuffer(2) = USB_STOP Then
```

```
                ApdUSBPanel.lstUSB.AddItem "EZUSB wrote back USB_STOP
recognizing request to stop dsp"
            End If
            ' Kill global variable
            'blnKill = True
            'ApdUSBPanel.lblKill.Caption = "Killed."
            '
        Else
            ApdUSBPanel.lstUSB.AddItem "Could not read a response from
device..."
        End If

    End If

    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

    If ApdOpt.chkLogAPDSAUSBToFile.value = 1 Then
        'close file
        Close #g_fNum
    End If

End Sub


'********************************************************************
'********************************************************************
Public Sub OpenLogFile()

' This opens and prepares, if selected in the APD options,
' a file that logs detailed data while it is collected.
' The filename comes from the global variable 'logfile'
' Calibration values, and data are put to file.

    Dim fNum As Integer
    Dim fname As String, fTitle As String

    'Set default directory
    ChDrive "C"
    ApdDir
    ChDir "C:\Apd"

    'Bring up Save dialog box.
    On Error GoTo EndOfSub3 'This is for when user clicks Cancel.

    g_fNum = FreeFile

    Open logFile For Output As #g_fNum
    Write #g_fNum, "Logged data..."
    Write #g_fNum, "Calibration Values"
    Write #g_fNum, "Press Offset", "Press Span", "Flow Offset", "Flow
Inh Span", "Flow Exh Span", "Duty Cycle"
    Write #g_fNum, "(Counts)", "(cmH20/count)", "(counts)",
"(Lps/count)", "(Lps/count)", "(counts 0x0400 = 1024 max)"
    Write #g_fNum, xp, ap, xo, ai, ae, duty
    Write #g_fNum, ""
    Write #g_fNum, "Respiratory Resistance (cmH20/Lps)", "Mouth
Pressure (counts)", "Flow (counts)"
```

```
    'Close #g_fNum
    'Leave file open for logging

    Exit Sub

EndOfSub3:
    MsgBox "Log file not selected."


End Sub



'******************************************************************
'******************************************************************
Public Sub StreamADC()

' This sub requests and then receives streamed ADC values from
' the APD-SA USB.  Streamed variables if received properly
' are then extracted from the stream and put into their
' respective buffers


'Stream ADC values from the device
Dim blnResult As Boolean
Dim intCount As Integer
Dim ReportsAttempted As Integer
Dim intVarsIndexLoc As Integer
Dim strLastFragment As String
Dim intTempTransferBufferCount As Integer
Dim intTempTransferBuffer() As Integer
Dim ReportsRead As Integer
Dim intWhenToStop As Integer

intVarsIndexLoc = 0
strLastFragment = ""

'global stuff for getting these vars etc.
ReDim Vars(1 To arraysize)
intVarsIndex = 0

'ReDim temp buffer according to needed temp buffer
intTempTransferBufferCount = 0
ReDim intTempTransferBuffer((CLng(USB_BYTES_PER_VAR) *
USB_MAX_STREAM_SIZE) - 1 + (3 * USB_BYTES_PER_REPORT))
'one integer slot to hold each
'added extra room for extra bytes in padded buffer at end for
'communication handshake etc. on stop
'The CLng in the above statement prevents overflow in array ReDim
'each digit of the streamed value
'ReDim intTempTransferBuffer((USB_BYTES_PER_VAR * 200) - 1)
'one integer slot to hold each


ReportsAttempted = 0
ReportsRead = 0
'Clear input report queue - Gets rid of the reports waiting in
'ring buffer
```

```
'that have not yet been read...
blnResult = HidD_FlushQueue(HID)
ApdUSBPanel.lstUSB.AddItem "Flush Queue Result: " & CStr(blnResult)
ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

'Assume the WriteFile function will set SendBuffer(0) = ReportID to
'zero
SendBuffer(1) = USB_STREAM_ADC
'this is global const
'
blnResult = DoAWriteFile()

If blnResult = True Then
    ApdUSBPanel.lstUSB.AddItem "Sent request to stream ADC values..."
    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
End If

'Read the ACK or NACK to this request
blnResult = ApdUsb.ReadSingleReport()
If blnResult = True Then
    If (ReadBuffer(1) = USB_DSP_ACK) Then
        blnResult = True
        With ApdUSBPanel.lstUSB
            .AddItem "Read ACK to stream ADC request."
            .AddItem "Now about to ReadSingleReport(s) containing
values."
            .AddItem "Screen here will only show the number of reports
read as they occur"
            .ListIndex = .ListCount - 1
        End With
    Else
        blnResult = False
    End If
End If

'*************
'Yes ACK was received so GET DATA streaming...
If blnResult = True Then

    'intWhenToStop = 20 '20 reports for debug and slow reporting
    intWhenToStop = CLng((CSng(USB_BYTES_PER_VAR) * USB_MAX_STREAM_SIZE
/ USB_BYTES_PER_REPORT) + 3)
    'this is about 657 reports for 7 seconds, 500 Hz, 2 channels, 6
     'bytes per var, 64 bytes per report
    'and requires about 93 reports per second, and thus 10ms per
     'report - which is well within bounds.
    'NOTE: the above value should provide at least one report beyond
     'the number of reports actually
    'containing data - e.g. is 656.25 reports required, get 658
    'For 5 seconds, its less but at the same rate 471 reports = 30144
     'bytes = 5024 values


    'Then during extract variables, only the needed variables are
     'extraced
    'The last report is just used to signal ACK of the STOP request
    'Use 3 report beyond fraction of needed for the manner in which
     'intWhenToStop-1 is checked and
    'then set to intWhenToStop at the end of the loop
```

```
    'intTempTransferBuffer needs to be appropriately large enough

    'Do loop until byte is returned indicating end of variables
     'transferred
    Do
        'Get report
        ReportsAttempted = ReportsAttempted + 1
        'ApdUSBPanel.lstUSB.AddItem "Reports Attempted = " & CStr
(ReportsAttempted) 'should show report number
        blnResult = (ReadSingleReport(False) > 0)
           'Calling with False means don't show read buffer
        'If blnResult = True Then
            'assign appropriate vars here
            'Print them as obtained
            'Shouldn't need to check for end of usb var transfer
                'here...because it's stopped from
            'host perspective
            '<!--
            'For intCount = 0 To UBound(ReadBuffer)
            '    If ReadBuffer(intCount) = USB_END_OF_VARIABLE_TRANSFER
Then
            '        ApdUSBPanel.lstUSB.AddItem "Found
USB_END_OF_VARIABLE_TRANSFER at: " _
            '            & CStr(intCount) & " (arrays are base 0);
value = " & Hex$(USB_END_OF_VARIABLE_TRANSFER)
            '        ApdUSBPanel.lstUSB.ListIndex =
ApdUSBPanel.lstUSB.ListCount - 1
            '        blnResult = False
            '    End If
            'Next intCount
            '-->
            'ApdUSBPanel.lstUSB.AddItem Hex$(ReadBuffer(1)) & ", " &
Hex$(ReadBuffer(2))
            'ApdUSBPanel.lstUSB.ListIndex =
ApdUSBPanel.lstUSB.ListCount - 1
        'End If
        '
        'If OK, then extract the variables from this buffer
        'Don't do this quite yet
        '<!--
        'strLastFragment = ExtractVariablesFromReadBuffer( _
        '    (Not (blnResult)), _
        '    strLastFragment, _
        '    intVarsIndexLoc)
        '-->
        '
        'Transfer each value to buffer for later processing
        If blnResult = True Then
            ReportsRead = ReportsRead + 1
            For intCount = 1 To UBound(ReadBuffer)
           'element 0 is report ID
                intTempTransferBuffer(intTempTransferBufferCount) =
ReadBuffer(intCount)
                intTempTransferBufferCount = intTempTransferBufferCount
+ 1
                If ReadBuffer(intCount) = USB_HOST_TOO_SLOW Then GoTo
ExitLoop_StreamADC
            Next intCount
        End If
```

```
        'Terminate the streaming of data if read sufficient number
           'of reports
        If ReportsRead = (intWhenToStop - 1) Then
'(USB_MAX_READ_REPORTS - 1) Then
            ApdUSBPanel.lstUSB.AddItem "Reached intWhenToStop; Calling
Stop DSP" 'USB_MAX_READ_REPORTS-1; Calling StopDsp"
            ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount
- 1
            Call StopDsp
            ReportsRead = intWhenToStop 'USB_MAX_READ_REPORTS
        End If

        DoEvents 'Hopefully this will allow for manual stop when
           'button is pressed on form...
        'may need to use instead the Kill global var and check with
           'each loop for its state

    Loop Until _
    ReportsAttempted = (intWhenToStop + USB_MAX_ATTEMPTED_REPORTS) _
    Or ReportsRead = intWhenToStop

ExitLoop_StreamADC:

    'Loop Until ReportsAttempted = USB_MAX_ATTEMPTED_REPORTS Or
     'ReportsRead = USB_MAX_READ_REPORTS 'ReadBuffer(1) =
     'USB_END_OF_VARIABLE_TRANSFER
    'Loop Until blnResult = False Or ReportsAttempted =
     'USB_MAX_READ_REPORTS 'ReadBuffer(1) =
     'USB_END_OF_VARIABLE_TRANSFER
    '
    'When finished collecting all data into transfer buffer, need to
     'extract the variables
    'ExtractVariables here !!!!!!!!!!!!!!!! <---<------
    'Need new extract variables based on the number of channels to
     'split and for this buffer
    blnResult = ExtractStreamedVariablesFromBuffer
(intTempTransferBuffer(), _
                CLng(USB_BYTES_PER_VAR) * USB_MAX_STREAM_SIZE - 1, _
                2)

    ' used to use intMaxIndex = CLng(USB_BYTES_PER_VAR) *
USB_MAX_STREAM_SIZE - 1

    With ApdUSBPanel.lstUSB
        .AddItem "intTempTransferBufferCount = " & CStr
(intTempTransferBufferCount)
        .ListIndex = .ListCount - 1
    End With

    'The with block below would be too long processing if kept in for
     'the full transfer block, so
    'just transfer the variables to the blocks
    'With ApdUSBPanel.lstStreamData
    '    For intCount = LBound(intTempTransferBuffer) To
(intTempTransferBufferCount - 1)
    '        .AddItem CStr(intTempTransferBuffer(intCount))
    '    Next intCount
    'End With
```

```
    With ApdUSBPanel.lstStreamData(0)
        .Clear
        For intCount = LBound(mp()) To UBound(mp())
             .AddItem CStr(mp(intCount))
        Next intCount
    End With
    With ApdUSBPanel.lstStreamData(1)
        .Clear
        For intCount = LBound(fr()) To UBound(fr())
            .AddItem CStr(fr(intCount))
        Next intCount
    End With

ElseIf blnResult = False Then
    ApdUSBPanel.lstUSB.AddItem "An ACK was not received after the
stream ADC request."
    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
End If

End Sub



'*********************************************************************
'*********************************************************************
Public Sub Run()

' This sub performs the 'run' function, that is it tells
' the APD-SA USB to begin collecting data and looking for perts
' while sending pressure, flow, and any RR data back to the host
' PC by USB.  This function includes the request of run mode, the
' receipt of acknowledge, and then the receipt of data, extraction
' of variables from the data stream.  If at any point, the
' the run needs to stop, either from user clicking stop button
' or from reaching max perts, then the sub uses stop dsp routine.


'Set APD USB to runmode

'Stream ADC values from the device
Dim blnResult As Boolean
Dim intCount As Integer
Dim ReportsAttempted As Integer
Dim intVarsIndexLoc As Integer
Dim strLastFragment As String
Dim intTempTransferBufferCount As Integer
Dim intTempTransferBuffer() As Integer
Dim ReportsRead As Integer
Dim intWhenToStop As Integer
Dim intInCount As Long
Dim intExCount As Long

'Prepare arrays and variables to hold the data.
inPertNo = 0
exPertNo = 0
inFftNo = 0
exFftNo = 0
intDisplayUpdateNumber = 0
```

```
intPertNo = 0 'to keep track of order of breaths, perts, to fill in
time of RR's (approx)

'ReDim inPI(0 To maxNumPerts - 1)
'Fix and zero max number of inh perturbations.
'ReDim exPI(0 To maxNumPerts - 1)
'Fix and zero max number of exh perturbations.
ReDim inPI(0 To maxNumPerts + 100)
'Padded for case of very uneven number of perts
ReDim exPI(0 To maxNumPerts + 100)
'Padded for case of very uneven number of perts
ReDim SI(1 To 1)
'Spectral analysis results dynamically allocated.

ReDim InShiftReg(0 To 0)
ReDim ExShiftReg(0 To 0)
intInCount = 0
intExCount = 0


intVarsIndexLoc = 0
strLastFragment = ""

ApdPanel.LogStatus.Caption = "USB Logging..."
'ApdPanel.Freq.Caption = "USB"
ApdUsb.ClearAPDMainDisplayUSB


'global stuff for getting these vars etc.
ReDim Vars(1 To arraysize)
intVarsIndex = 0

'ReDim temp buffer according to needed temp buffer
intTempTransferBufferCount = 0
ReDim intTempTransferBuffer((CLng(USB_BYTES_PER_VAR) *
USB_MAX_RUN_SIZE) - 1 + (3 * USB_BYTES_PER_REPORT))
'one integer slot to hold each
'added extra room for extra bytes in padded buffer at end for
'communication handshake etc. on stop
'The CLng in the above statement prevents overflow in array ReDim
'each digit of the streamed value
'ReDim intTempTransferBuffer((USB_BYTES_PER_VAR * 200) - 1)
'one integer slot to hold each


ReportsAttempted = 0
ReportsRead = 0
'Clear input report queue - Gets rid of the reports waiting in ring
'buffer
'that have not yet been read...
blnResult = HidD_FlushQueue(HID)
ApdUSBPanel.lstUSB.AddItem "Flush Queue Result: " & CStr(blnResult)
ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

'Assume the WriteFile function will set SendBuffer(0) = ReportID to
'zero
SendBuffer(1) = USB_RUN 'this is global const
'
blnResult = DoAWriteFile()
```

```
If blnResult = True Then
    ApdUSBPanel.lstUSB.AddItem "Sent request to run..."
    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
End If

'Read the ACK or NACK to this request
blnResult = ApdUsb.ReadSingleReport()
If blnResult = True Then
    If (ReadBuffer(1) = USB_DSP_ACK) Then
        blnResult = True
        With ApdUSBPanel.lstUSB
            .AddItem "Read ACK to run request."
            .AddItem "Now about to ReadSingleReport(s) containing
values."
            .AddItem "Screen here will only show the number of reports
read as they occur"
            .ListIndex = .ListCount - 1
        End With

        blnKill = False
        ApdUSBPanel.lblKill.Caption = "Not Killed."
        '

    Else
        blnResult = False
    End If
End If

'*************
'Yes ACK was received so GET DATA streaming...
If blnResult = True Then

    'intWhenToStop = 20 '20 reports for debug and slow reporting
    'intWhenToStop = CLng((CSng(USB_BYTES_PER_VAR) *
     'USB_MAX_RUN_SIZE / USB_BYTES_PER_REPORT))
    intWhenToStop = 180
    'the above = 187.5 reports - but need number of reports divisible
     'by 3 to be an int.
    'thus use 180 reports which is about a second but is 1920 complete
     'variable values
    'the above value should be something like 1 second x 500Hz x 4
     'variables x 6 bytes per var
    ' divided by 64 bytes per report and should give an integer before
     'conversion to long
    'for example in the present case: 12,000 bytes = 187.5 reports.
    'thus use a multiple of 3 reports so that complete variables are
     'extracted since 10 2/3
    'variables are returned with each report (3 reports gives exactly
     '32 variables)
Begin_of_Run_Loop:
    ReportsAttempted = 0
    ReportsRead = 0
    intTempTransferBufferCount = 0

    'Do loop until proper amount of data is collected for processing
    Do
        'Get report
        ReportsAttempted = ReportsAttempted + 1
```

```
        'ApdUSBPanel.lstUSB.AddItem "Reports Attempted = " & CStr
(ReportsAttempted) 'should show report number
        blnResult = (ReadSingleReport(False) > 0)
     'Calling with False means don't show read buffer

        'Transfer each value to buffer for later processing
        If blnResult = True Then
            ReportsRead = ReportsRead + 1
            For intCount = 1 To UBound(ReadBuffer)
                'element 0 is report ID
                intTempTransferBuffer(intTempTransferBufferCount) =
ReadBuffer(intCount)
                intTempTransferBufferCount = intTempTransferBufferCount
+ 1
                If ReadBuffer(intCount) = USB_HOST_TOO_SLOW Then GoTo
ExitLoop_Run
            Next intCount
        End If

        'Terminate the streaming of data if read sufficient number of
           'reports
        If ReportsRead = (intWhenToStop) Then
           'Report number is base 1
            ApdUSBPanel.lstUSB.AddItem "Reached intWhenToStop;
transferring data" 'USB_MAX_READ_REPORTS-1; Calling StopDsp"
            ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount
- 1
            'Call StopDsp 'nope - for run mode, just tranfer data and
                'keep collecting until stop is requested by user or
                'max data reached
            ReportsRead = intWhenToStop 'USB_MAX_READ_REPORTS
        End If

        DoEvents
           'Hopefully this will allow for manual stop when button is
           'pressed on form...
           'may need to use instead the Kill global var and check with
           'each loop for its state

    Loop Until _
    ReportsAttempted = (intWhenToStop + USB_MAX_ATTEMPTED_REPORTS) _
    Or ReportsRead = intWhenToStop _
    Or blnKill = True

ExitLoop_Run:

    If blnKill = True Then
        Call StopDsp
        GoTo End_of_Run
    End If

    'When finished collecting all data into transfer buffer, need to
     'extract the variables
    'ExtractVariables here !!!!!!!!!!!!!!!! <---<------
    'Need new extract variables based on the number of channels to
     'split and for this buffer
    'blnResult = ExtractStreamedVariablesFromBuffer
(intTempTransferBuffer(), _
    '             CLng(USB_BYTES_PER_VAR) * USB_MAX_RUN_SIZE - 1, _
```

```
    '            USB_RUN_VARS)
    blnResult = ExtractStreamedVariablesFromBuffer
(intTempTransferBuffer(), _
                CLng(USB_BYTES_PER_REPORT) * intWhenToStop - 1, _
                USB_RUN_VARS)

    'if error in transfer then restart the read loop...
    If blnResult = False Then
        GoTo Begin_of_Run_Loop
    End If

'Process variable values - perturbations etc.
Call ProcessPertsUSB(CInt(intWhenToStop * USB_BYTES_PER_REPORT /
USB_BYTES_PER_VAR / USB_RUN_VARS))

'Update the display if necessary
Call UpdateDisplayUSB

'If logging data, write to file
'Call LogDataUSB
'Or data may be saved with the usual file save method.

'If reached max num of desired perts then
If inPertNo >= maxNumPerts + 10 And exPertNo >= maxNumPerts + 10 Then
    With ApdUSBPanel.lstUSB
        .AddItem "Reached maxNumPerts for both inPertNo and exPertNo"
        .ListIndex = .ListCount - 1
    End With
    Call StopDsp
    blnKill = True
    MsgBox "Reached maximum number of inhalation and exhalation
perturbations."
    UpdateDisplayUSB (True) 'Update display with average of all values.
End If

'If continue reading reports / data then goto beginning of loop again
to refill buffer
If blnKill = False Then
    GoTo Begin_of_Run_Loop
End If

End_of_Run:


    With ApdUSBPanel.lstUSB
        .AddItem "intTempTransferBufferCount = " & CStr
(intTempTransferBufferCount)
        .ListIndex = .ListCount - 1
    End With

    With ApdUSBPanel.lstStreamData(0)
        For intCount = LBound(mp()) To UBound(mp())
            .AddItem CStr(mp(intCount))
        Next intCount
    End With
    With ApdUSBPanel.lstStreamData(1)
        For intCount = LBound(fr()) To UBound(fr())
            .AddItem CStr(fr(intCount))
        Next intCount
```

```
    End With

ElseIf blnResult = False Then
    ApdUSBPanel.lstUSB.AddItem "An ACK was not received after the run
request."
    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
End If

ApdPanel.LogStatus.Caption = "Not Logging"

End Sub


'*******************************************************************
'*******************************************************************
Public Sub UpdateDisplayUSB(Optional AverageAll As Boolean = False)

' This sub updates the ApdPanel display with USB-derived RR data.
' Based on early fast-reponse device simulations, this function
' still includes the ability to use a filter for updating the display
' or for using the simulated analog display.
' This is now buffered pert data - about one second's worth of data
' updating
' thus need to look through data to find how many perts are there

Static InShiftReg() As PertInfo
Static ExShiftReg() As PertInfo
Static intInCount, intExCount, intLBnd, intUBnd As Integer
Dim blnIsExh As Boolean
Dim PertValue As PertInfo
Dim TempShift() As PertInfo
Dim intCount As Integer
Dim oldUBound As Integer

Dim cnt As Integer
Dim inR() As Single, exR() As Single 'Resistance.
Dim inFR() As Single, exFR() As Single 'Flowrate.
Dim inVFR() As Single, exVFR() As Single 'Flow reduction.
Dim inDev() As Single, exDev() As Single 'Device resistance.
Dim period() As Single 'Period from one perturbtion to the next.
Dim inA As Single, exA As Single 'Temporary variables.

If intDisplayUpdateNumber = 0 Then
    ReDim InShiftReg(0 To 0)
    ReDim ExShiftReg(0 To 0)
    intInCount = 0
    intExCount = 0
End If


If AverageAll = True Then

    ApdPanel.StatusBar.Caption = "All value averaged - to max. no.
specified in options."

    'Take average of all freq estimates.
    exA = UBound(SI()) + 1 'add one more element to SI array
    ReDim Preserve SI(LBound(SI()) To exA)
    SI(exA).f1 = 0 'zero it to be sure
```

```
    For intCount = (LBound(SI()) + 1) To (UBound(SI()) - 1)
    'Go from LBound + 1 because due to flow of program, first element
     'is always 0
        SI(exA).f1 = SI(exA).f1 + SI(intCount).f1
           'add all prior elements
    Next intCount

    SI(exA).f1 = SI(exA).f1 / CSng(UBound(SI()) - LBound(SI()))
     'divide to acheive avg

    'Write new average value to display
    inA = SI(UBound(SI())).f1
    ApdPanel.Freq.Caption = Format(inA, "0.0") ' + " Hz"


    '*****************************************************
    '** INH data averages
    PertValue.R = 0
    PertValue.afr = 0
    PertValue.vfr = 0

    For intCount = 0 To inPertNo - 1 'maxNumPerts - 1
     'Sum all values...
        'inR(intCount) = inPI(intCount + intInCount).R
        'inFR(intCount) = inPI(intCount + intInCount).afr
        'inVFR(intCount) = inPI(intCount + intInCount).vfr
        PertValue.R = PertValue.R + inPI(intCount).R
        PertValue.afr = PertValue.afr + inPI(intCount).afr
        PertValue.vfr = PertValue.vfr + inPI(intCount).vfr
    Next intCount

    'Divide values to get average
    PertValue.R = PertValue.R / CSng(inPertNo - 1) '(maxNumPerts)
    PertValue.afr = PertValue.afr / CSng(inPertNo - 1) '(maxNumPerts)
    PertValue.vfr = PertValue.vfr / CSng(inPertNo - 1) '(maxNumPerts)

    blnIsExh = False

    'Might consider removing large data points here and msg to user
     'indicating such:
    'calc mean and std dev and then any pts larger than mean + 5 sd

    Call ApdUsb.UpdateAPDMainUSBDisplay(PertValue, intInCount,
intExCount, blnIsExh)

    '*****************************************************************
    '**EXH Now...
    PertValue.R = 0
    PertValue.afr = 0
    PertValue.vfr = 0


    For intCount = 0 To exPertNo - 1 'maxNumPerts - 1 'Sum all values
        PertValue.R = PertValue.R + exPI(intCount).R
        PertValue.afr = PertValue.afr + exPI(intCount).afr
        PertValue.vfr = PertValue.vfr + exPI(intCount).vfr
    Next intCount
```

```
    'Divide values to get averages
    PertValue.R = PertValue.R / CSng(exPertNo - 1) '(maxNumPerts)
    PertValue.afr = PertValue.afr / CSng(exPertNo - 1) '(maxNumPerts)
    PertValue.vfr = PertValue.vfr / CSng(exPertNo - 1) '(maxNumPerts)

    blnIsExh = True

    Call ApdUsb.UpdateAPDMainUSBDisplay(PertValue, intInCount,
intExCount, blnIsExh)



    GoTo UDU_End

End If


'** If not using filter because filter length = 1
'this is really just average of each period
'and is thus an average of data collection interval
If APDSimSettings.sldFilterLength.value = 1 Then
    intDisplayUpdateNumber = intDisplayUpdateNumber + 1
    'ApdMain.UpdateDisplay (False)
     'Use APD main update display data to show data collected
    ApdPanel.StatusBar.Caption = "Unfiltered perturbation data."
    ApdPanel.StatusBar2.Caption = " "

    'ReDim inR(0 To inPertNo - intInCount - 1)
    'ReDim exR(0 To exPertNo - intExCount - 1)
    'ReDim inFR(0 To inPertNo - intInCount - 1)
    'ReDim exFR(0 To exPertNo - intExCount - 1)
    'ReDim inVFR(0 To inPertNo - intInCount - 1)
    'ReDim exVFR(0 To exPertNo - intExCount - 1)

    '*******************************************
    '* INHALATION
    '*******************************************

    If inPertNo < 1 Then GoTo Check_Exh_Perts_No_Shift
     'if no inh perts yet
    If inPertNo - intInCount = 0 Then GoTo Check_Exh_Perts_No_Shift
     'if no new perts since last cycle

    PertValue.R = 0
    PertValue.afr = 0
    PertValue.vfr = 0

    For intCount = 0 To inPertNo - intInCount - 1 'Sum all values...
        'inR(intCount) = inPI(intCount + intInCount).R
        'inFR(intCount) = inPI(intCount + intInCount).afr
        'inVFR(intCount) = inPI(intCount + intInCount).vfr
        PertValue.R = PertValue.R + inPI(intCount + intInCount).R
        PertValue.afr = PertValue.afr + inPI(intCount + intInCount).afr
        PertValue.vfr = PertValue.vfr + inPI(intCount + intInCount).vfr
    Next intCount

    'Divide values to get average
    PertValue.R = PertValue.R / CSng(inPertNo - intInCount)
    PertValue.afr = PertValue.afr / CSng(inPertNo - intInCount)
```

```
    PertValue.vfr = PertValue.vfr / CSng(inPertNo - intInCount)

    blnIsExh = False

    Call ApdUsb.UpdateAPDMainUSBDisplay(PertValue, intInCount,
intExCount, blnIsExh)

    intInCount = inPertNo


    '*****************************************
    '* EXHALATION
    '*****************************************
Check_Exh_Perts_No_Shift:

    If exPertNo < 1 Then GoTo UDU_End
    If exPertNo - intExCount = 0 Then GoTo UDU_End

    PertValue.R = 0
    PertValue.afr = 0
    PertValue.vfr = 0


    For intCount = 0 To exPertNo - intExCount - 1 'Sum all values
        'exR(intCount) = exPI(intCount + intExCount).R
        'exFR(intCount) = exPI(intCount + intExCount).afr
        'exVFR(intCount) = exPI(intCount + intExCount).vfr
        PertValue.R = PertValue.R + exPI(intCount + intExCount).R
        PertValue.afr = PertValue.afr + exPI(intCount + intExCount).afr
        PertValue.vfr = PertValue.vfr + exPI(intCount + intExCount).vfr
    Next intCount

    'Divide values to get averages
    PertValue.R = PertValue.R / CSng(exPertNo - intExCount)
    PertValue.afr = PertValue.afr / CSng(exPertNo - intExCount)
    PertValue.vfr = PertValue.vfr / CSng(exPertNo - intExCount)

    blnIsExh = True

    Call ApdUsb.UpdateAPDMainUSBDisplay(PertValue, intInCount,
intExCount, blnIsExh)

    intExCount = exPertNo


'** If yes, use filter (filter length > 1) then...
ElseIf APDSimSettings.sldFilterLength.value > 1 Then

    intDisplayUpdateNumber = intDisplayUpdateNumber + 1

    ApdPanel.StatusBar.Caption = "Filtered perturbation data."
    ApdPanel.StatusBar2.Caption = " "

    If inPertNo < 1 Then GoTo Check_Exh_Perts 'if no inh perts yet
    If inPertNo - intInCount = 0 Then GoTo Check_Exh_Perts 'if no new
perts since last cycle

    ' If inh filter not yet full because not yet enough data to
     'fill it...
```

```
    If (inPertNo - 1) < APDSimSettings.sldFilterLength.value Then
        intLBnd = 0
        intUBnd = inPertNo - 1

        ReDim Preserve InShiftReg(intLBnd To intUBnd)

        For intCount = 0 To inPertNo - 1
            InShiftReg(intCount) = inPI(intCount)
        Next intCount

        PertValue = ApdSim.AvgFilter(InShiftReg)

        blnIsExh = False

        Call ApdUsb.UpdateAPDMainUSBDisplay(PertValue, intInCount,
intExCount, blnIsExh)

        ApdPanel.StatusBar.Caption = "Inh filter not full"

        intInCount = inPertNo

    '** If inh filter is full then
    ElseIf (inPertNo - 1) >= APDSimSettings.sldFilterLength.value Then

        blnIsExh = False '** This is inhalation not exhalation

        oldUBound = UBound(InShiftReg)

        intLBnd = 0

        intUBnd = UBound(InShiftReg) + (inPertNo - intInCount)

        '** Allocate new empty space(s) in Inh Shift Register
        ReDim Preserve InShiftReg(intLBnd To intUBnd)

        '** Place new pert value into the Inh Shift Register
        For intCount = 1 To (inPertNo - intInCount)
            InShiftReg(oldUBound + intCount) _
            = inPI(intInCount + intCount - 1)
        Next intCount

        '** Clear and redim the temporary shift register
        ReDim TempShift(0 To APDSimSettings.sldFilterLength.value - 1)

        '** Dump the last elements of Inh Shift Register into Temp Reg
        Call ShiftArrayUSB(InShiftReg, TempShift, inPertNo -
intInCount)

        '** Clear and redim Inh Shift Register to max size
        ReDim InShiftReg(0 To APDSimSettings.sldFilterLength.value - 1)

        '** Dump the temp stored values into Inh Shift Register
        Call ShiftArrayUSB(TempShift, InShiftReg, 0)

        '** Send the Inh Shift Register to be filtered to one value
        PertValue = ApdSim.AvgFilter(InShiftReg)

        '** Send the value to the display
```

```
        Call ApdUsb.UpdateAPDMainUSBDisplay(PertValue, intInCount,
intExCount, blnIsExh)

        '** Update ApdPanel status bar
        ApdPanel.StatusBar.Caption = "Inh filter filled."

        '** Increment count of inhalation perts processed
        intInCount = inPertNo

    End If

Check_Exh_Perts:

    If exPertNo < 1 Then GoTo UDU_End
    If exPertNo - intExCount = 0 Then GoTo UDU_End

    '** If Exhalation filter not yet full then...
    If (exPertNo - 1) < APDSimSettings.sldFilterLength.value Then
        blnIsExh = True '** This is exhalation pert
        intLBnd = 0
        intUBnd = exPertNo - 1

        '** Add next element to Exh Shift Register
        ReDim Preserve ExShiftReg(intLBnd To intUBnd)

        For intCount = 0 To exPertNo - 1
            ExShiftReg(intCount) = exPI(intCount)
        Next intCount

        PertValue = ApdSim.AvgFilter(ExShiftReg)

        Call ApdUsb.UpdateAPDMainUSBDisplay(PertValue, intInCount,
intExCount, blnIsExh)

        ApdPanel.StatusBar.Caption = "Exh filter not full"

        intExCount = exPertNo

    '** If exhalation filter is full then
    ElseIf (exPertNo - 1) >= APDSimSettings.sldFilterLength.value Then
        blnIsExh = True '** This is exhalation not inhalation

        oldUBound = UBound(ExShiftReg)

        intLBnd = 0

        intUBnd = UBound(ExShiftReg) + (exPertNo - intExCount)

        '** Allocate new empty space(s) in Exh Shift Register
        ReDim Preserve ExShiftReg(intLBnd To intUBnd)

        '** Place new pert value into the Exh Shift Register
        For intCount = 1 To (exPertNo - intExCount)
            ExShiftReg(oldUBound + intCount) _
            = exPI(intExCount + intCount - 1)
        Next intCount 'exPI is base 0

        '** Clear and redim the temporary shift register
        ReDim TempShift(0 To APDSimSettings.sldFilterLength.value - 1)
```

```
        '** Dump the last elements of Exh Shift Register into Temp Reg
        Call ShiftArrayUSB(ExShiftReg, TempShift, exPertNo -
intExCount)

        '** Clear and redim Exh Shift Register to max size
        ReDim ExShiftReg(0 To APDSimSettings.sldFilterLength.value - 1)

        '** Dump the temp stored values into Exh Shift Register
        Call ShiftArrayUSB(TempShift, ExShiftReg, 0)

        '** Send the Exh Shift Register to be filtered to one value
        PertValue = ApdSim.AvgFilter(ExShiftReg)

        '** Send the value to the display
        Call ApdUsb.UpdateAPDMainUSBDisplay(PertValue, intInCount,
intExCount, blnIsExh)

        '** Update ApdPanel status bar
        ApdPanel.StatusBar.Caption = "Exh filter filled."

        '** Increment count of inhalation perts processed
        intExCount = exPertNo

    End If

End If

'** If analog indicators are used, then update them
'** Use average resp resist value if it's displayed on main panel yet
ApdMain.UpdateIndicators ("Main")

UDU_End:

End Sub
Sub ClearAPDMainDisplayUSB()

ApdPanel.Rin.Caption = "?"
ApdPanel.Rex.Caption = "?"
ApdPanel.Rav.Caption = "?"
ApdPanel.FRin.Caption = "?"
ApdPanel.FRex.Caption = "?"
ApdPanel.FRav.Caption = "?"
ApdPanel.FPin.Caption = "?"
ApdPanel.FPex.Caption = "?"
ApdPanel.FPav.Caption = "?"
ApdPanel.Pin.Caption = Str(0)
ApdPanel.Pex.Caption = Str(0)
ApdPanel.Freq = "?.?"
ApdPanel.lblMode.Caption = "USB"

End Sub

Sub UpdateAPDMainUSBDisplay(PertData As PertInfo, ByVal inPertNo As
Integer, _
ByVal exPertNo As Integer, blnIsExh As Boolean)

'** This sub updates the main APD panel display with USB data
'** it uses the main APD Panel
```

```
'** timer.

Dim inFP, exFP, inA, exA As Single
Const minperts As Integer = 3

If inPertNo < minperts And _
exPertNo < minperts Then
        ApdPanel.Rin.Caption = "?"
        ApdPanel.Rex.Caption = "?"
        ApdPanel.Rav.Caption = "?"
        ApdPanel.FRin.Caption = "?"
        ApdPanel.FRex.Caption = "?"
        ApdPanel.FRav.Caption = "?"
        ApdPanel.FPin.Caption = "?"
        ApdPanel.FPex.Caption = "?"
        ApdPanel.FPav.Caption = "?"
        ApdPanel.Pin.Caption = Str(inPertNo)
        ApdPanel.Pex.Caption = Str(exPertNo)
        Exit Sub
End If


'** Display updated resistance values
'** Display updated flow rates
'** Display updated percent flow perturbation
'** Display perturbation numbers

If blnIsExh = True Then '** Update exhalation if it's exhalation pert
    With ApdPanel
        If exPertNo < minperts Then
            .Rex.Caption = "?"
            .Rav.Caption = "?"
            .FRex.Caption = "?"
            .FRav.Caption = "?"
            .FPex.Caption = "?"
            .FPav.Caption = "?"
            .Pin.Caption = Str(inPertNo)
            .Pex.Caption = Str(exPertNo)
            Exit Sub
        Else
            .Rex.Caption = Format(PertData.R, "0.00")
            .FRex.Caption = Format(Abs(PertData.vfr * ae), "0.00")
            If PertData.vfr = 0 Then
                exFP = 0
            Else
                exFP = (1 - PertData.afr / PertData.vfr) * 100#
            End If
            .FPex.Caption = Format(exFP, "0.0")
            .Pex.Caption = Str(exPertNo)
        End If
    End With
ElseIf blnIsExh = False Then
    With ApdPanel
        If inPertNo < minperts Then
            .Rin.Caption = "?"
            .Rav.Caption = "?"
            .FRin.Caption = "?"
            .FRav.Caption = "?"
            .FPin.Caption = "?"
```

```
            .FPav.Caption = "?"
            .Pin.Caption = Str(inPertNo)
            .Pex.Caption = Str(exPertNo)
            Exit Sub
        Else
            .Rin.Caption = Format(PertData.R, "0.00")
            .FRin.Caption = Format(Abs(PertData.vfr * ai), "0.00")
            If PertData.vfr = 0 Then
                inFP = 0
            Else
                inFP = (1 - PertData.afr / PertData.vfr) * 100#
            End If
            .FPin.Caption = Format(inFP, "0.0")
            .Pin.Caption = Str(inPertNo)
        End If
    End With
End If

'** Calculate averages if both in and ex values are not "?"
With ApdPanel
    If IsNumeric(.Rin.Caption) And IsNumeric(.Rex.Caption) Then
        .Rav.Caption = Format(0.5 * (CSng(.Rin.Caption) + CSng
(.Rex.Caption)), "0.00")
    End If
    If IsNumeric(.FRin.Caption) And IsNumeric(.FRex.Caption) Then
        .FRav.Caption = Format(0.5 * (Abs(CSng(.FRin.Caption)) + Abs
(CSng(.FRex.Caption))), "0.00")
    End If
    If IsNumeric(.FPin.Caption) And IsNumeric(.FPex.Caption) Then
        .FPav.Caption = Format(0.5 * (CSng(.FPin.Caption) + CSng
(.FPex.Caption)), "0.0")
    End If
End With

'** Display frequency - this cannot be reconstructed from current
'** APD saved pert files so it's set to 12.3 Hz
'inA = 12.3
'ApdPanel.Freq.Caption = "USB" 'Format(inA, "0.0") + " Hz"
If UBound(SI()) > 2 And SI(UBound(SI())).f1 >= Val(ApdOpt.txtMinFreq)
Then  'only update is some values have been calculated
    ApdPanel.Freq.Caption = Format(SI(UBound(SI())).f1, "0.0")
End If

End Sub
Sub ShiftArrayUSB(inarray() As PertInfo, SubArray() As PertInfo, _
intOffset As Integer)

'** This function outputs ShiftArray, a subset of InArray,
'offset by intOffset

Dim intCount As Integer

For intCount = LBound(SubArray) To UBound(SubArray)
    SubArray(UBound(SubArray) - intCount + LBound(SubArray)) _
    = inarray(UBound(inarray) - intCount + LBound(inarray))
Next intCount

End Sub
```

```
'*******************************************************************
'*******************************************************************
Public Sub ProcessPertsUSB(intMaxIndex As Integer)

' This sub processes perts from rd buffer just transferred
' from the ExtractVariables
' routine.  It looks for pert value in the buffer, fills pert
' info structures etc.  The pert info structs are used for
' display updating, file writing etc. and is also the way
' that pert information and flow and pressure at the time of
' perturbation is retained while data buffers are re-written.

Dim intCount As Integer
Dim n As Integer
Dim intLastCount As Integer
Dim intLastPertDir As Integer
Const EXH As Integer = 1
Const INH As Integer = 2
Const NOTFOUNDYET As Integer = -1
'Const MAXACCEPTABLETIMEDIFFERENCE As Integer = 50
'500 Hz sample rate, max pert freq:
'Const MINACCEPTABLETIMEDIFFERENCE As Integer = 10
'minimum samples
Dim MAXACCEPTABLETIMEDIFFERENCE As Integer
Dim MINACCEPTABLETIMEDIFFERENCE As Integer
Dim sngFreqs() As Single
Dim intFreqCount As Integer

'Inits
ReDim sngFreqs(1 To intMaxIndex)
intLastCount = 0
'for estimating frequency by monitoring samples between perts.
intLastPertDir = NOTFOUNDYET
intFreqCount = 0 'zero freqs estimated to begin with
MAXACCEPTABLETIMEDIFFERENCE = Int(OURSCANRATE / 2# / CSng(Val
(ApdOpt.txtMinFreq)))  '500 samples per second / 7 Hz
MINACCEPTABLETIMEDIFFERENCE = Int(OURSCANRATE / 2# / CSng(Val
(ApdOpt.txtMaxFreq)))  '500 samples per second / 14 Hz
'Divided by two because runmode is 250 Hz to be able to stream data

'Loop through this buffer looking for perts and assigning to inh or
'exh based on flow direction
'In processing of data, normally, assign pressure to pert.amp (actual
'mouth pressure) and flow to pert.afr (actual flow rate)
'If a pert is present in this USB mode, then the RR value in the sets
'of 3 data points sent by the APD-SA USB
'will not be zero, and thus the virtual and actual flow rates are sent
'- thus data is assigned to the next pert
'accordingly.
For intCount = LBound(rd) To intMaxIndex - (1 - LBound(rd))
    'look for values <>0 to indicate the presence of a pert value
    'then transfer pert value, press and flow values into pert info
structures

    'log data to file if this option is selected
    If ApdOpt.chkLogAPDSAUSBToFile.value = 1 Then
        Write #g_fNum, rd(intCount), mp(intCount), fr(intCount)
```

```
    End If

    If rd(intCount) <> 0 Then 'pert found
        'intLastCount = intLastCount + 1
           'increment counter for perts found for freq. det. - may not
           'need extra counter here
        intPertNo = intPertNo + 1 'increment counter of pert numbers
        If negIsExh = True Then
            'Debug.Print "negIsExh=True"
            If fr(intCount) > 0 Then
                'If inPertNo >= maxNumPerts Then GoTo SkipPert
                If inPertNo > UBound(inPI) Then GoTo SkipPert
                 'prevent overrunning end of array
                inPI(inPertNo).afr = fr(intCount)
                If ApdOpt.chkAPDSASendsAFRVFRatRR.value = 1 Then
                    inPI(inPertNo).vfr = mp(intCount)
                      'if this option is set, the APD-SA actually
                    'sends vfr in the mp place only at the point of an
                      'RR val
                Else
                    inPI(inPertNo).amp = mp(intCount)
                End If
                inPI(inPertNo).R = rd(intCount)
                inPI(inPertNo).abstime = intPertNo

                'Estimate frequency based on last consecutive good
                 'pert in same direction
                If intLastCount > 1 Then 'This really doesn't matter
                      'now: if not the first pert found in this data
                      'set
                    'Debug.Print "Inh: intLastCount > 1"
                    If intLastPertDir = INH Then
                      'if already found one and it was INH direction
                        'Debug.Print "Inh: intLastPertDir=INH"
                        Debug.Print "Inh: intCount-intLastCount: " &
CStr(intCount - intLastCount)
                        If ((intCount - intLastCount) <
MAXACCEPTABLETIMEDIFFERENCE) _
                        And ((intCount - intLastCount) >
MINACCEPTABLETIMEDIFFERENCE) Then
                        'If there was not too much space between last
                           'pert and this pert in the same direction
                        'i.e. these are most likely consecutive perts
                       'Debug.Print "It is less than and greater than"
                            intFreqCount = intFreqCount + 1
                           'I miss programming in C
                            sngFreqs(intFreqCount) = 1# / (CSng
(intCount - intLastCount) * CSng(DT * 2#))  'Hz = 1/(samples*DT)
                            Debug.Print CStr(sngFreqs(intFreqCount)) &
"Is sngFreqs(intFreqCOunt)"
                            'mult denom by 2 because 250 Hz run mode
                        End If
                    End If
                End If

                intLastPertDir = INH

                inPertNo = inPertNo + 1
```

```
            ElseIf fr(intCount) < 0 Then
                'If exPertNo >= maxNumPerts Then GoTo SkipPert
                If exPertNo > UBound(exPI) Then GoTo SkipPert
                exPI(exPertNo).afr = fr(intCount)
                If ApdOpt.chkAPDSASendsAFRVFRatRR.value = 1 Then
                    exPI(exPertNo).vfr = mp(intCount)
                      'if this option is set, the APD-SA actually
                      'sends vfr in the mp place only at the point of
                      'an RR val
                Else
                    exPI(exPertNo).amp = mp(intCount)
                End If
                exPI(exPertNo).R = rd(intCount)
                exPI(exPertNo).abstime = intPertNo

                'Estimate frequency based on last consecutive
                 'good pert in same direction
                If intLastCount > 1 Then
                      'This really doesn't matter now: if not the
                      'first pert found in this data set
                    If intLastPertDir = EXH Then
                      'if already found one and it was INH direction
                        If ((intCount - intLastCount) <
MAXACCEPTABLETIMEDIFFERENCE) _
                        And ((intCount - intLastCount) >
MINACCEPTABLETIMEDIFFERENCE) Then
                      'If there was not too much space between last
                      'pert and this pert in the same direction
                        'i.e. these are most likely consecutive perts
                            intFreqCount = intFreqCount + 1
                           'I miss programming in C
                            sngFreqs(intFreqCount) = 1# / (CSng
(intCount - intLastCount) * CSng(DT * 2#))  'Hz = 1/(samples*DT)
                        End If
                    End If
                End If

                intLastPertDir = EXH

                exPertNo = exPertNo + 1
            End If
        End If
        If negIsExh = False Then
            'Debug.Print "negIsExh = False"
            If fr(intCount) < 0 Then
                'If inPertNo >= maxNumPerts Then GoTo SkipPert
                If inPertNo > UBound(inPI) Then GoTo SkipPert
                inPI(inPertNo).afr = fr(intCount)
                If ApdOpt.chkAPDSASendsAFRVFRatRR.value = 1 Then
                    inPI(inPertNo).vfr = mp(intCount)
                      'if this option is set, the APD-SA actually
                      'sends vfr in the mp place only at the point of
                      'an RR val
                Else
                    inPI(inPertNo).amp = mp(intCount)
                End If
                inPI(inPertNo).R = rd(intCount)
                inPI(inPertNo).abstime = intPertNo
```

```
                'Estimate frequency based on last consecutive
                 'good pert in same direction
                If intLastCount > 1 Then
                      'This really doesn't matter now: if not the
                      'first pert found in this data set
                    If intLastPertDir = INH Then
                      'if already found one and it was INH direction
                        If ((intCount - intLastCount) <
MAXACCEPTABLETIMEDIFFERENCE) _
                        And ((intCount - intLastCount) >
MINACCEPTABLETIMEDIFFERENCE) Then
                        'If there was not too much space between
                           'last pert and this pert in the same
                           'direction
                        'i.e. these are most likely consecutive perts
                            intFreqCount = intFreqCount + 1
                               'I miss programming in C
                            sngFreqs(intFreqCount) = 1# / (CSng
(intCount - intLastCount) * CSng(DT * 2#)) 'Hz = 1/(samples*DT)
                        End If
                    End If
                End If

                intLastPertDir = INH

                inPertNo = inPertNo + 1

            ElseIf fr(intCount) > 0 Then
                'If exPertNo >= maxNumPerts Then GoTo SkipPert
                If exPertNo > UBound(exPI) Then GoTo SkipPert
                exPI(exPertNo).afr = fr(intCount)
                If ApdOpt.chkAPDSASendsAFRVFRatRR.value = 1 Then
                    exPI(exPertNo).vfr = mp(intCount)
                      'if this option is set, the APD-SA actually
                      'sends vfr in the mp place only at the point of
                      'an RR val
                Else
                    exPI(exPertNo).amp = mp(intCount)
                End If
                exPI(exPertNo).R = rd(intCount)
                exPI(exPertNo).abstime = intPertNo

                'Estimate frequency based on last consecutive good
                 'pert in same direction
                If intLastCount > 1 Then
                      'This really doesn't matter now: if not the
                      'first pert found in this data set
                    If intLastPertDir = EXH Then
                      'if already found one and it was INH direction
                        If ((intCount - intLastCount) <
MAXACCEPTABLETIMEDIFFERENCE) _
                        And ((intCount - intLastCount) >
MINACCEPTABLETIMEDIFFERENCE) Then
                        'If there was not too much space between last
                           'pert and this pert in the same direction
                        'i.e. these are most likely consecutive perts
                            intFreqCount = intFreqCount + 1
                           'I miss programming in C
```

```
                            sngFreqs(intFreqCount) = 1# / (CSng
(intCount - intLastCount) * CSng(DT * 2#)) 'Hz = 1/(samples*DT)
                        End If
                    End If
                End If

                intLastPertDir = EXH

                exPertNo = exPertNo + 1
            End If
        End If
        intLastCount = intCount
           'for estimating freq. - save last position of pert
    End If 'if rd<>0 -> i.e. pert found

SkipPert:

Next intCount

'Now that this round of perts (Or lack thereof is read,
'update the shift registers for the
'display averaging - do this in update display.

'if found some frequencies and no doing FFTs
'Comment out FFT provisions until these are implemented
If (intFreqCount > 0) Then 'And Not (ApdOpt.dDoFFT.value = 1) Then
    'Augment the SI array and place in the new slot the
     'average present freq estimates
    n = UBound(SI()) + 1
    ReDim Preserve SI(LBound(SI()) To n)
    SI(n).f1 = 0 'zero this to be sure
    For intCount = 1 To intFreqCount
        SI(n).f1 = SI(n).f1 + sngFreqs(intCount)
           'add freqs to this value
    Next intCount
    SI(n).f1 = SI(n).f1 / intFreqCount 'average over the number found
    'Now when display is updated, the upper most SI.f1 value
     'will contain the latest
    'freq estimation for this batch of data.  Otherwise,
     'SI.f1 of latest value
    'will be from FFT
    Debug.Print "intFreqCount: " & CStr(intFreqCount) & "; intFreqs
(intFreqCount): " & CStr(sngFreqs(intFreqCount))

End If

'If (intFreqCount > 0) And (ApdOpt.dDoFFT.value = 1) Then
'    'If fft selected, do it.
'    If ApdOpt.dDoFFT.value = 1 Then
'        ApdMain.SpectralAnalysis 'this will be called and
'          the sub should now check to
'        'see if USB is in use according APD menu checked items and
'then proceed accordingly
'    End If
'End If

End Sub
```

```
'****************************************************************
'****************************************************************
Public Sub Connect()

' This sub coordinates all the routines to connect to the APD-SA
' USB.  It goes through the paces of finding the Cypress device,
' re-numerating after downloading firmware, and then upon final
' connection to the renumerated device, calibration constants are
' downloaded to the host PC.  If this sub doesn't find the
' appropriate device, then errors are displayed, and USB functions
' are aborted.

    On Error Resume Next
     'if error finding hid and downloading, just go to next
     'command, the error should show up and this will just exit

    Dim blnResult As Boolean
    Dim intTwiddle As Integer
    Dim blnCTOK As Boolean
    Dim NumBuffers As Long

    NumBuffers = USB_NUM_INPUT_BUFFERS
     '32 buffers instead of the standard 8

    With ApdUSBConnectPanel
        .lstAPDUSBConnect.AddItem "Looking for attached APD-SA USB
Device..."
        .lstAPDUSBConnect.ListIndex = .lstAPDUSBConnect.ListCount - 1
    End With

    'Connect to USB interface
    blnResult = FindTheHid()

    'If didn't find the device, try downloading the
     'firmware...just try once
    If blnResult = False Then

        With ApdUSBConnectPanel
            .lstAPDUSBConnect.AddItem "Device not found."
            .lstAPDUSBConnect.AddItem "Attempting to download firmware
to device / APD-SA..."
            .lstAPDUSBConnect.ListIndex = .lstAPDUSBConnect.ListCount -
1
        End With

        Call Download

        ApdUSBPanel.lstUSB.AddItem "Download attempted. Waiting for OS
Driver load...)"
        ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

        With ApdUSBConnectPanel
            .lstAPDUSBConnect.AddItem "Waiting for Windows to Load USB
Driver..."
            .lstAPDUSBConnect.ListIndex = .lstAPDUSBConnect.ListCount -
1
        End With

        'Now wait for re-numeration and OS to load proper
```

```
            'HID driver...
        'This will only work if the driver has been loaded before
        ApdUSBPanel.tmrTimout.Enabled = True
        Do
            intTwiddle = intTwiddle + 1
            DoEvents 'allow the OS to do its stuff
        Loop Until ApdUSBPanel.tmrTimout.Enabled = False
        blnResult = FindTheHid()
            'try one more time after the so called download
    End If

    If blnResult = True Then
        ApdUSBPanel.lstUSB.AddItem "Connected to APD USB Interface"
        ApdUSBPanel.lstUSB.AddItem "Now looking for connection to the
APD Brain itself " & vbCrLf _
                            & "(not just its USB mouth)"
        ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

        With ApdUSBConnectPanel
            .lstAPDUSBConnect.AddItem "Connected to APD-SA USB
Interface."
            .lstAPDUSBConnect.AddItem "Attempting to communicate with
APD-SA..."
            .lstAPDUSBConnect.ListIndex = .lstAPDUSBConnect.ListCount -
1
        End With

        'Now since found, redim the data buffers
        If Capabilities.OutputReportByteLength > 0 Then
            ReDim SendBuffer(Capabilities.OutputReportByteLength - 1)
        Else
            ReDim SendBuffer(0)
            ApdUSBPanel.lstUSB.AddItem "OutputReportByteLength=0??????"
        End If
        If Capabilities.InputReportByteLength > 0 Then
            ReDim ReadBuffer(Capabilities.InputReportByteLength - 1)
        Else
            ReDim ReadBuffer(0)
            ApdUSBPanel.lstUSB.AddItem "InputReportByteLength=0??????"
        End If
    Else
        ApdUSBPanel.lstUSB.AddItem "Cannot connect to USB Interface"
        With ApdUSBConnectPanel
            .lstAPDUSBConnect.AddItem "Cannot connect to APD-SA USB
Interface."
            .lstAPDUSBConnect.AddItem "Check your USB cable connection,
try unplugging the device,"
            .lstAPDUSBConnect.AddItem "cycling power to the APD-SA and
replugging the USB cable."
            .lstAPDUSBConnect.ListIndex = .lstAPDUSBConnect.ListCount -
1
        End With
    End If



    'If OK, then DSP connect using the USB interface
    If blnResult = True Then
           'Clear input report queue - Gets rid of the reports
```

```
           'waiting in ring buffer
        'that have not yet been read...
        blnResult = HidD_FlushQueue(HID)
        ApdUSBPanel.lstUSB.AddItem "Flush Queue Result: " & CStr
(blnResult)
        ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
        '
        blnResult = HidD_SetNumInputBuffers(HID, NumBuffers)
        ApdUSBPanel.lstUSB.AddItem "HidD_SetNumInputBuffers: " & CStr
(blnResult)
        ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
        'Oops, this isn't included in the
           'kernel32.dll!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        '
        'Assume the WriteFile function will set SendBuffer(0) =
           'ReportID to zero
        SendBuffer(1) = USB_CONNECT_TO_DSP 'USB_UNRECOGNIZED_COMMAND
           'USB_CONNECT_TO_DSP 'this is global const
        '
        blnResult = DoAWriteFile()
        '
        If blnResult = True Then
           'wrote file OK, so check for DSP ACK connection etc.
            ApdUSBPanel.lstUSB.AddItem "Wrote File OK. Waiting for
acknowledgement from DSP."
            '
            'Do ReadFile waiting for ACK
            blnResult = (ReadSingleReport() > 0)
                'ReadSingleReport returns the number of bytes read
            '
            'If...then write ok...connected to USB APD
            If blnResult = True Then
                ApdUSBPanel.lstUSB.AddItem "Read something here..."
                ApdUSBPanel.lstUSB.AddItem "Result: " & Hex$(ReadBuffer
(1)) & ", outcount: " & Hex$(ReadBuffer(2))
           'assuming (0) is report ID ?
                If ReadBuffer(1) = USB_DSP_NO_ACK Then
                    ApdUSBPanel.lstUSB.AddItem "=
USB_DSP_NO_ACK...either timeout on connect or"
                    ApdUSBPanel.lstUSB.AddItem "connect reject...
ouch."
                End If
                If ReadBuffer(1) = USB_DSP_ACK Then
                    ApdUSBPanel.lstUSB.AddItem "Looks like we're
connected people."
                    With ApdUSBConnectPanel
                        .lstAPDUSBConnect.AddItem "Communcation to APD-
SA Ok."
                        .lstAPDUSBConnect.AddItem "Success: Connected
to APD-SA."
                        .lstAPDUSBConnect.AddItem " "
                        .lstAPDUSBConnect.AddItem "Requesting APD-SA
calibration variables..."
                        .lstAPDUSBConnect.ListIndex = .
lstAPDUSBConnect.ListCount - 1
                    End With

                    ApdUSBPanel.cmdRequestVariables.Enabled = True
                    ApdUSBPanel.cmdStreamADC.Enabled = True
```

```
                    'Check for commtimeouts stuff - no dice - maybe
                      'only for serial
                    'keep getting invalid handle messages...probably
                      'cuz not handle to serial
                    'blnCTOK = GetCommTimeouts(HID,
                      'CurrentCommTimeout)
                    'ApdUSBPanel.lstUSB.AddItem "GetCommTimeouts
                      'result: " & CStr(blnCTOK)
                    'ApdUSBPanel.lstUSB.AddItem Cstr
                      '(CurrentCommTimeout.ReadTotalTimeoutConstant)
                    'ApdUSBPanel.lstUSB.AddItem Cstr
                      '(CurrentCommTimeout.ReadTotalTimeoutMultiplier)
                    'ApdUSBPanel.lstUSB.ListIndex =
                      'ApdUSBPanel.lstUSB.ListCount - 1
                    'Call DisplayResultOfAPICall("GetCommTimeouts")
                    ApdUSBPanel.lstUSB.AddItem "Requesting SAAPD
variables over USB..."
                    ApdUsb.RequestVariables

                    ApdPanel.lblMode.Caption = "USB"

                End If
            Else
                ApdUSBPanel.lstUSB.AddItem "Could not read a response
from device..."
                ApdUSBPanel.lstUSB.AddItem "Not connected to APD
Brain."
                With ApdUSBConnectPanel
                    .lstAPDUSBConnect.AddItem "Cannot communicate with
APD-SA."
                    .lstAPDUSBConnect.AddItem "Not connected to APD
(just USB Interface)."
                    .lstAPDUSBConnect.AddItem "Try unplugging the
device,"
                    .lstAPDUSBConnect.AddItem "cycling power to the
APD-SA and replugging the USB cable."
                    .lstAPDUSBConnect.ListIndex = .
lstAPDUSBConnect.ListCount - 1
                End With
                'Here need to get rid of HID connection  maybe?
            End If

        ElseIf blnResult = False Then
            With ApdUSBConnectPanel
                .lstAPDUSBConnect.AddItem "Could not write USB data or
communicate with APD-SA."
                .lstAPDUSBConnect.AddItem "Try quitting this program,
unplugging the device to reset the Windows USB HID Driver,"
                .lstAPDUSBConnect.AddItem "cycling power to the APD-SA
and replugging the USB cable."
                .lstAPDUSBConnect.ListIndex = .
lstAPDUSBConnect.ListCount - 1
            End With
        End If
    End If

    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

    ApdUSBConnectPanel.cmdOK.Enabled = True
```

```
    If blnResult = False Then
           'Get rid of any handle to USB communications if not
           'properly connected.
        Call ApdUsb.ShutdownUSB
    End If

End Sub


'*********************************************************************
'*********************************************************************
Public Sub ShowCalDefaultsUSB()

' This sub loads into the calibration window typical default
' (theoretical) calibration constants for an APD-SA USB with
' characteristics that are shown in the popup window.
' This could be expanded to offer several alternatives or
' even a library of calibration sets.

    With ApdCal

        MsgBox "Values are for typical" & vbCrLf _
        & "+/-1 inH20 DC001NDR5 flow sensor " & vbCrLf _
        & "+/-10 inH20 DC010NDR5 pressure sensor " & vbCrLf _
        & "with ADS7825 ADC chip (16-bit). Sensors are 0.25 to 4.25 VDC
output "

        .txtDuty = CStr(256)
        'default pressure cal vals
        .Span1 = CStr(-0.003876) 'cmH20 / adc count
        .Offset1.Text = CStr(7373) 'adc counts
        'default flow cal vals
        .SpanI.Text = CStr(-0.001252) 'lps/adc count
        .SpanE.Text = CStr(-0.001252) 'lps/adc count
        .Offset0.Text = CStr(7373)

        .dNegIsExh.value = 1

    End With


End Sub


'*********************************************************************
'*********************************************************************
Public Sub GetHostVars()

' Though called "GetHostVars" this is actually to transfer variables
' calculated by this host PC to the target device.
' Variables are transferred to the APD-SA USB and then written to
' FLASH.  Variables are calibration variables.

Dim blnResult As Boolean
Dim intCount As Integer
Dim intBufIndex As Integer

intBufIndex = 1
```

```
'Clear input report queue - Gets rid of the reports waiting in ring
'buffer
'that have not yet been read...
blnResult = HidD_FlushQueue(HID)
ApdUSBPanel.lstUSB.AddItem "Flush Queue Result: " & CStr(blnResult)
ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

'Signal write host vars to DSP
SendBuffer(intBufIndex) = USB_GET_HOST_VARS

'Write each vars as bytes to the sendbuffer
' Note that after each OutDecFloat, intBufIndex has already been
incremented to the next index
intBufIndex = intBufIndex + 1
With ApdUSBPanel.lstGetHostVars
      .Clear
      .AddItem CStr(ApdMain.xp)
      Call OutDecFloat(ApdMain.xp, intBufIndex) 'pressure offset
      'Write digits to window too for debugging purposes
      For intCount = LBound(Digits) To UBound(Digits)
            ApdUSBPanel.lstGetHostVars.AddItem CStr(Digits(intCount))
      Next intCount
      .AddItem CStr(ApdMain.xo)
      Call OutDecFloat(ApdMain.xo, intBufIndex) 'flow offset
      'Write digits to window too for debugging purposes
      For intCount = LBound(Digits) To UBound(Digits)
            ApdUSBPanel.lstGetHostVars.AddItem CStr(Digits(intCount))
      Next intCount
      .AddItem CStr(ApdMain.ap)
      Call OutDecFloat(1# / ApdMain.ap, intBufIndex) 'pressure span inh
      'inverted during send preserve sig. digs.
      'Write digits to window too for debugging purposes
      For intCount = LBound(Digits) To UBound(Digits)
            ApdUSBPanel.lstGetHostVars.AddItem CStr(Digits(intCount))
      Next intCount
      .AddItem CStr(ApdMain.ai)
      Call OutDecFloat(1# / ApdMain.ai, intBufIndex) 'flow span inh
      'inverted during send preserve sig. digs.
      'Write digits to window too for debugging purposes
      For intCount = LBound(Digits) To UBound(Digits)
            ApdUSBPanel.lstGetHostVars.AddItem CStr(Digits(intCount))
      Next intCount
      .AddItem CStr(ApdMain.duty)
      Call OutDecFloat(CSng(ApdMain.duty), intBufIndex) 'duty cycle
      'Write digits to window too for debugging purposes
      For intCount = LBound(Digits) To UBound(Digits)
            ApdUSBPanel.lstGetHostVars.AddItem CStr(Digits(intCount))
      Next intCount
End With

'Signal end of variables
SendBuffer(intBufIndex) = USB_END_OF_VARIABLE_TRANSFER 'already
incremented in the OutDecFloat routine

'Send buffer to EZUSB
blnResult = DoAWriteFile()

If blnResult = True Then
```

```
    ApdUSBPanel.lstUSB.AddItem "Sent request to write host vars
(USB_GET_HOST_VARS)..."
    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
End If

'Read the ACK or NACK to this request
blnResult = ApdUsb.ReadSingleReport()
If blnResult = True Then
    If (ReadBuffer(1) = USB_DSP_ACK) Then
        blnResult = True
        With ApdUSBPanel.lstUSB
            .AddItem "Read ACK to GET_HOST_VARS request."
            '.AddItem "Now about to ReadSingleReport containing
variables."
            .ListIndex = .ListCount - 1
        End With
    Else
        blnResult = False
    End If
End If

'Yes ACK was received then do something if you want...?
If blnResult = True Then

ElseIf blnResult = False Then
    ApdUSBPanel.lstUSB.AddItem "An ACK was not received after the
GET_HOST_VARS request."
    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
End If

End Sub


'*******************************************************************
'*******************************************************************
Public Sub RequestVariables()

' This sub sends a byte to the DSP that says,
' "please send back all necessary variables"
' It then recieves the variables (calibration variables)

Dim blnResult As Boolean
Dim intCount As Integer
Dim ReportsAttempted As Integer
Dim intVarsIndexLoc As Integer
Dim strLastFragment As String

intVarsIndexLoc = 0
strLastFragment = ""

'global stuff for getting these vars etc.
ReDim Vars(0 To (USB_NUM_VARS - 1))
intVarsIndex = 0

ReportsAttempted = 0

'Clear input report queue - Gets rid of the reports waiting
'in ring buffer
```

```
'that have not yet been read...
blnResult = HidD_FlushQueue(HID)
ApdUSBPanel.lstUSB.AddItem "Flush Queue Result: " & CStr(blnResult)
ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

'Assume the WriteFile function will set SendBuffer(0) = ReportID
'to zero
SendBuffer(1) = USB_REQUEST_VARIABLES 'USB_UNRECOGNIZED_COMMAND
'USB_CONNECT_TO_DSP 'this is global const
'
blnResult = DoAWriteFile()

If blnResult = True Then
    ApdUSBPanel.lstUSB.AddItem "Sent request for all relevent
variables..."
    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
End If

'Read the ACK or NACK to this request
blnResult = ApdUsb.ReadSingleReport()
If blnResult = True Then
    If (ReadBuffer(1) = USB_DSP_ACK) Then
        blnResult = True
        With ApdUSBPanel.lstUSB
            .AddItem "Read ACK to send vars request."
            .AddItem "Now about to ReadSingleReport containing
variables."
            .ListIndex = .ListCount - 1
        End With
    Else
        blnResult = False
    End If
End If
'Yes ACK was received
If blnResult = True Then
    'Do loop until byte is returned indicating end of
     'variables transferred
    Do
        'Get report
        ReportsAttempted = ReportsAttempted + 1
        Debug.Print "REPORTS ATTEMPTED: " & CStr(ReportsAttempted)
        blnResult = (ReadSingleReport() > 0)
        If blnResult = True Then
            'assign appropriate vars here
            'Print them as obtained
            For intCount = 0 To UBound(ReadBuffer)
                If ReadBuffer(intCount) = USB_END_OF_VARIABLE_TRANSFER
Then
                    ApdUSBPanel.lstUSB.AddItem "Found
USB_END_OF_VARIABLE_TRANSFER at: " _
                        & CStr(intCount) & " (arrays are base 0); value
= " & Hex$(USB_END_OF_VARIABLE_TRANSFER)
                    ApdUSBPanel.lstUSB.ListIndex =
ApdUSBPanel.lstUSB.ListCount - 1
                    blnResult = False
                End If
            Next intCount
            'ApdUSBPanel.lstUSB.AddItem Hex$(ReadBuffer(1)) & ", " &
                'Hex$(ReadBuffer(2))
```

```
            'ApdUSBPanel.lstUSB.ListIndex =
ApdUSBPanel.lstUSB.ListCount - 1
        End If
        '
        'If OK, then extract the variables from this buffer
        strLastFragment = ExtractVariablesFromReadBuffer( _
            (Not (blnResult)), _
            strLastFragment, _
            intVarsIndexLoc)
        DoEvents
    Loop Until _
    blnResult = False _
    Or ReportsAttempted = USB_MAX_READ_REPORTS _
    Or blnKill = True 'ReadBuffer(1) = USB_END_OF_VARIABLE_TRANSFER

    ApdMain.xp = Vars(0) 'pressoff
    ApdMain.xo = Vars(1) 'flowoff
    ApdMain.ap = 1# / Vars(2)  'pressspan
     'inverted during send preserve sig. digs.
    ApdMain.ai = 1# / Vars(3)  'flowspan inh
     'inverted during send preserve sig. digs.
    ApdMain.ae = 1# / Vars(3) 'flowspan exh
     'inverted during send preserve sig. digs.
    ApdMain.duty = Vars(4) 'duty cycle


    ApdUSBPanel.lstVars.Clear
    Vars(2) = 1# / Vars(2)
    Vars(3) = 1# / Vars(3)
    For intCount = LBound(Vars) To UBound(Vars)
        ApdUSBPanel.lstVars.AddItem CStr(Vars(intCount))
    Next intCount

    With ApdUSBConnectPanel
        .lstAPDUSBConnect.AddItem "Received variables from APD-SA USB."
        .lstAPDUSBConnect.ListIndex = .lstAPDUSBConnect.ListCount - 1
    End With

ElseIf blnResult = False Then
    ApdUSBPanel.lstUSB.AddItem "An ACK was not received after the send
variable request."
    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

    With ApdUSBConnectPanel
        .lstAPDUSBConnect.AddItem "Did not properly receive variables
after the send variable request."
        .lstAPDUSBConnect.ListIndex = .lstAPDUSBConnect.ListCount - 1
    End With


End If

End Sub


'*******************************************************************
'*******************************************************************
Private Function ExtractVariablesFromReadBuffer( _
    blnFoundEndOfVars As Boolean, _
```

```
    strLastFragment As String, _
    ByRef intVarsIndexLoc As Integer _
    ) As String


' This sub does the following:
' Extract the variables from the buffer based on the number of digits
' expected.
' Adjust later for number of buffer, multi-buffer use, etc. including
' placement
' of call to this function...
'
' Returns: the remaining string of the digits in the buffer that were
' not
' enough to form a complete character i.e. if there are only 4 of 6
' digits
' remaining in the buffer, the four are returned, and the next 2 are
' assumed to arise in the next buffer read...


Dim intBufferIndex, intDigitNum, intDig As Integer
'intVarNum As Integer
'Dim Vars() As Single
Dim strVal As String
Dim strLine As String

'ReDim Vars(USB_MAX_VARS_PER_BUFFER)
'This will create 11 slots for USB...=10
'intVarNum = 0

'Start at 1 because (0) is record number...
'''For intBufferIndex = 1 To (UBound(ReadBuffer) - USB_BYTES_PER_VAR +
1)
'intBufferIndex = 1 + Len(strLastFragment)

intBufferIndex = 1
' nope-> + Len(strLastFragment)  ....that's wrong, no need to jump
' ahead there buddy old pal, mkay?
strVal = strLastFragment

Do

    For intDigitNum = 0 To (USB_BYTES_PER_VAR - 1 - Len(strVal))
        'If reading fragment of digit and past end of buffer then exit
        If (intBufferIndex + intDigitNum) > UBound(ReadBuffer) Then
GoTo ExitLoop
        'If reading an end of var transmit value, then exit cuz last
        'good digit was already saved.
        intDig = ReadBuffer(intBufferIndex + intDigitNum)
        If intDig = USB_END_OF_VARIABLE_TRANSFER Then GoTo ExitLoop
        'Ignore large numbers (from codes, etc., prepare for reading
           'other digit codes too
        If intDig <= 9 Then
           'hopefully not confused with negatives here...
            strVal = strVal & CStr(intDig)
        ElseIf intDig = USB_NEG_SIGN Then
            strVal = strVal & "-"
        ElseIf intDig = USB_DEC_PT Then
            strVal = strVal & "."
```

```
        ElseIf intDig > &HAF Then
           '45 and 46 are ascii symbols, acks, nacks etc are >0xb_
            If ApdUSBPanel.chkShowHexCodes.value = True Then
                strVal = strVal & Hex$(intDig)
            Else
                strVal = strVal & "0"
            End If
        Else
            strVal = strVal & "0"
        End If
    Next intDigitNum
    Vars(intVarsIndexLoc) = CSng(strVal)
    intBufferIndex = intBufferIndex + intDigitNum 'USB_BYTES_PER_VAR
    intVarsIndexLoc = intVarsIndexLoc + 1
    'Just for DEBUG - will need to be adjusted
    If intVarsIndexLoc > UBound(Vars) Then
        intVarsIndexLoc = 0
    End If

    'MAY need to check for running past the end of this Vars array....
    strVal = ""

Loop Until intBufferIndex > UBound(ReadBuffer)

ExitLoop:
ExtractVariablesFromReadBuffer = strVal 'save the last fragment if
there is one


'For intBufferIndex = 0 To UBound(Vars)
'    ApdUSBPanel.lstUSB.AddItem CStr(Vars(intBufferIndex))
'    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
'Next intBufferIndex

'Dim intCount As Integer
'Dim strLine As String

With ApdUSBPanel.lstUSB

    .AddItem "Vars contents:"
    .ListIndex = .ListCount - 1

    For intBufferIndex = 0 To UBound(Vars)
        If (Len(CStr(Vars(intBufferIndex))) >= 6) Then
            strLine = strLine & Hex$(intBufferIndex) & ": " & CStr(Vars
(intBufferIndex)) & Chr(9) 'this is tab
        Else
            strLine = strLine & Hex$(intBufferIndex) & ": " & CStr(Vars
(intBufferIndex)) & Chr(9) & Chr(9) 'this is tab
        End If
        If ((intBufferIndex + 1) Mod 8 = 0) Then 'if its a multiple of
8 thensesx
            'strLine = strLine & Chr(10) & Chr(13)
                'linefeed and carriage return ASCII
            .AddItem strLine
            .ListIndex = .ListCount - 1
            strLine = ""
        End If
    Next intBufferIndex
```

```
    'Add last line if not multiple of 16
    .AddItem strLine
    .ListIndex = .ListCount - 1

End With

End Function


'*******************************************************************
'*******************************************************************
Private Function ExtractStreamedVariablesFromBuffer( _
    ByRef intBuffer() As Integer, _
    intMaxIndex As Long, _
    intNumberOfSets As Integer _
    ) As Boolean

' This sub extracts pressure and flow or pressure, flow and RR
' variables from a data stream from the USB.
'
'Assumes that for whatever number of data set in the buffer, that the
'order is always:
'pressure, flow, respiratory resistance, some other debug integer type
'of data here.
'
'Extract the variables from the buffer based on the number of digits
'expected.
'Adjust later for number of buffer, multi-buffer use, etc. including
'placement
'of call to this function...

Dim intBufferIndex, intDigitNum, intDig As Integer
Dim strVal As String
Dim strLine As String
Dim intVarsIndexLoc As Long
Dim intVarSetNum As Integer

On Error GoTo ESVFB_ErrorHandler

intBufferIndex = 0 'incoming buufer is base 0
' nope-> + Len(strLastFragment)  ....that's wrong, no need to jump
'ahead there buddy old pal, mkay?
strVal = ""
intVarsIndexLoc = 0 'reset counter for destination of vars
intVarSetNum = 1

Do

    For intDigitNum = 0 To (USB_BYTES_PER_VAR - 1 - Len(strVal))
        'If reading fragment of digit and past end of buffer then exit
        If (intBufferIndex + intDigitNum) > intMaxIndex Then GoTo
ExitLoop
        'If reading an end of var transmit value, then exit cuz last
        'good digit was already saved.
        intDig = intBuffer(intBufferIndex + intDigitNum)
        With ApdUSBPanel.lstUSB
            If intDig = USB_END_OF_VARIABLE_TRANSFER Then
                .AddItem "Found USB_END_OF_VARIABLE_TRANSFER"
```

```
                .ListIndex = .ListCount - 1
                GoTo ExitLoop
            End If
            If intDig = USB_STOP Then
                .AddItem "Found USB_STOP"
                .ListIndex = .ListCount - 1
                GoTo ExitLoop
            End If
            If intDig = USB_HOST_TOO_SLOW Then
                .AddItem "Found USB_HOST_TOO_SLOW"
                .ListIndex = .ListCount - 1
                GoTo ExitLoop
            End If
        End With

        'Ignore large numbers (from codes, etc., prepare for reading
           'other digit codes too
        If intDig <= 9 Then
           'hopefully not confused with negatives here...
            strVal = strVal & CStr(intDig)
        ElseIf intDig = USB_NEG_SIGN Then
            strVal = strVal & "-"
        ElseIf intDig = USB_DEC_PT Then
            strVal = strVal & "."
        ElseIf intDig > &HAF Then '45 and 46 are ascii symbols, acks,
           'nacks etc are >0xb_
            If ApdUSBPanel.chkShowHexCodes.value = True Then
                strVal = strVal & Hex$(intDig)
            Else
                strVal = strVal & "0"
            End If
        Else
            strVal = strVal & "0"
        End If
    Next intDigitNum

    If CSng(strVal > 32000) Then
        'Put msgbox here.

        'MsgBox "Error: ExtractStreamedVariablesFromBuffer: Bad data
           'transfer from DSP to USB to HOST. " _
        '& vbCrLf & "Please try again."
        Debug.Print "Too large extracted streamed var - raising err 13"
        err.Raise 13
    End If

    'Check cases here for number of data sets in the stream and split
     'according to arrays.
    Select Case intNumberOfSets
        Case 1
            Vars(intVarsIndexLoc) = CSng(strVal)
            'Just for DEBUG - will need to be adjusted
            If intVarsIndexLoc > UBound(Vars) Then
                intVarsIndexLoc = 0
            End If

        Case 2 'pressure and flow data in the stream of data
            'first value is pressure & second is flow
            If intVarSetNum = 1 Then
```

```
                mp(intVarsIndexLoc + 1) = CSng(strVal)
                 'mp, fr are base 1
            ElseIf intVarSetNum = 2 Then
                fr(intVarsIndexLoc + 1) = CSng(strVal)
            End If
        Case USB_RUN_VARS
           'data is pressure, flow at an rr value with extra field -
           'say number of pert or similar
            If intVarSetNum = 1 Then
                mp(intVarsIndexLoc + 1) = CSng(strVal)
                 'mp, fr are base 1
            ElseIf intVarSetNum = 2 Then
                fr(intVarsIndexLoc + 1) = CSng(strVal)
            ElseIf intVarSetNum = 3 Then
                rd(intVarsIndexLoc + 1) = CSng(strVal)
                If Abs(rd(intVarsIndexLoc + 1)) < USB_ERR_PAD Then
                    rd(intVarsIndexLoc + 1) = 0#
                      'within tolerance for precision errors on dsp
                End If
            'ElseIf intVarSetNum = 4 Then
                 'presently in this run mode, only 3 vars streamed
            '    Vars(intVarsIndexLoc + 1) = CSng(strVal)
            End If
        Case Else
            'ExtractStreamedVariablesFromBuffer = False
                 'Failed to have proper input value
            'GoTo ExitLoop
    End Select


    'intVarsIndexLoc = 0
     ' Make sure to reset and increment this as needed
    'Vars(intVarsIndexLoc) = CSng(strVal)
    intBufferIndex = intBufferIndex + intDigitNum 'USB_BYTES_PER_VAR
    'intVarsIndexLoc = intVarsIndexLoc + 1
     'index in destination variable

    intVarSetNum = intVarSetNum + 1 'increment set for next variable
    If intVarSetNum > intNumberOfSets Then
        intVarSetNum = 1 'reset variable set number to first variable
           'is exceeded the number of sets
        intVarsIndexLoc = intVarsIndexLoc + 1
           'index in destination variable
    End If

    'MAY need to check for running past the end of this Vars array....
    strVal = ""

Loop Until intBufferIndex > intMaxIndex

ExitLoop:
ExtractStreamedVariablesFromBuffer = True
'save the last fragment if there is one
ApdUSBPanel.lstUSB.AddItem "intVarsIndexLoc on exit ESVFB: " & CStr
(intVarsIndexLoc)
ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1


'For intBufferIndex = 0 To UBound(Vars)
```

```
'    ApdUSBPanel.lstUSB.AddItem CStr(Vars(intBufferIndex))
'    ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
'Next intBufferIndex

''Dim intCount As Integer
''Dim strLine As String

GoTo ESVFB_End

ESVFB_ErrorHandler:
If err.Number = 13 Then
    Call UpdateErrorStatusUSB("ExtractStreamedVariablesFromBuffer",
err.Number)
    ExtractStreamedVariablesFromBuffer = False
Else
    MsgBox "Error in ESVFB: " & CStr(err.Number) & " " &
err.Description
End If

ESVFB_End:

End Function


'*******************************************************************
'*******************************************************************
Public Sub UpdateErrorStatusUSB(Optional strDescrip As String = " ", _
                    Optional lngNum As Long = 0)

' This sub sends error codes to the USB panel - usually used when
' a mangled USB variable transfer has been detected - this is
' detected when a variable is far outside of its expected range.
' This type of range exceedence occurs when bit or byte have dropped
' or swapped, or during a stalled transfer.

With ApdUSBPanel.lstUSB
    .AddItem "Error: " & strDescrip
    .AddItem CStr(lngNum)
    .ListIndex = .ListCount - 1
End With

ApdPanel.StatusBar2.Caption = "Error in USB Data transfer - reset
suggested."

End Sub


'*******************************************************************
'*******************************************************************
Function ShowVarsContents()

' This sub is primarily for debugging and prints to the USB test
' panel variables that have been transferred.

Dim intBufferIndex As Integer
Dim strLine As String

With ApdUSBPanel.lstUSB
```

```
    .AddItem "Vars contents:"
    .ListIndex = .ListCount - 1

    For intBufferIndex = 0 To UBound(Vars)
        If (Len(CStr(Vars(intBufferIndex))) >= 6) Then
            strLine = strLine & Hex$(intBufferIndex) & ": " & CStr(Vars
(intBufferIndex)) & Chr(9) 'this is tab
        Else
            strLine = strLine & Hex$(intBufferIndex) & ": " & CStr(Vars
(intBufferIndex)) & Chr(9) & Chr(9) 'this is tab
        End If
        If ((intBufferIndex + 1) Mod 8 = 0) Then
           'if its a multiple of 8 thensesx
            'strLine = strLine & Chr(10) & Chr(13)
           'linefeed and carriage return ASCII
            .AddItem strLine
            .ListIndex = .ListCount - 1
            strLine = ""
        End If
    Next intBufferIndex

    'Add last line if not multiple of 16
    .AddItem strLine
    .ListIndex = .ListCount - 1

End With

End Function


'*******************************************************************
'*******************************************************************
Private Function DoAWriteFile() As Boolean

'Attempts a WriteFile call. Fails if the device isn't attached.

Dim Count As Integer
Dim NumberOfBytesRead As Long
Dim NumberOfBytesToSend As Long
Dim NumberOfBytesWritten As Long
Dim result2 As Integer
Dim result As Integer

'** Commented out so that data can be setup in other calls to this
'function
'The SendBuffer array begins at 0, so subtract 1 from the number of
'bytes.
'If Capabilities.OutputReportByteLength > 0 Then
'    ReDim SendBuffer(Capabilities.OutputReportByteLength - 1)
'Else
'    ReDim SendBuffer(0)
'End If

'*******************************************************************
'WriteFile
'Sends a report to the device.
'Returns: success or failure.
'Requires: the handle returned by CreateFile and
'The output report byte length returned by HidP_GetCaps
```

```
'******************************************************************

'The first byte is the Report ID
SendBuffer(0) = 0

'This is commented out to allow other functions to set data and call
'DoAWriteFile to
'output the data over USB
'The next bytes are data
'For Count = 1 To Capabilities.OutputReportByteLength - 1
'    SendBuffer(Count) = OutputReportData(Count - 1)
'Next Count

NumberOfBytesWritten = 0

result = WriteFile _
    (HID, _
    SendBuffer(0), _
    CLng(Capabilities.OutputReportByteLength), _
    NumberOfBytesWritten, _
    HIDOverlapped)
     'Used to be: 0) - HIDOverlapped is unsupported in the API
Call DisplayResultOfAPICall("WriteFile")

'Wait for ReadFile to complete or a timeout whichever comes first at
'spec'ed timeout interval
result2 = WaitForSingleObject _
    (EventObject, _
    1000) '1 second (# of ms)

Call DisplayResultOfAPICall("WaitForSingleObject")
'NOTE: even if this timed out, the printed result from the above
'display result
'will give "completed properly"

ApdUSBPanel.lstUSB.AddItem " OutputReportByteLength = " &
Capabilities.OutputReportByteLength
ApdUSBPanel.lstUSB.AddItem " NumberOfBytesWritten = " &
NumberOfBytesWritten
ApdUSBPanel.lstUSB.AddItem " Report ID: " & SendBuffer(0)
ApdUSBPanel.lstUSB.AddItem " Output Report Data:"

If result2 = WAIT_TIMEOUT Or result2 = WAIT_ABANDONED Then
    DoAWriteFile = False
    ApdUSBPanel.lstUSB.AddItem " WriteFile Timed Out"
ElseIf result2 = WAIT_OBJECT_0 Then
    DoAWriteFile = True
    ShowSendBuffer
End If

'For Count = 1 To UBound(SendBuffer)
'    ApdUSBPanel.lstUSB.AddItem " " & Hex$(SendBuffer(Count))
'Next Count

'Scroll to the bottom of the list box.
ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

End Function
```

```
'******************************************************************
'******************************************************************
Private Function FindTheHid() As Boolean
'Makes a series of API calls to locate the desired HID-class device.
'Returns True if the device is detected, False if not detected.

Dim Count As Integer
Dim GUIDString As String
Dim HidGuid As GUID
Dim MemberIndex As Long


LastDevice = False
MyDeviceDetected = False

'*******************************************************************
'HidD_GetHidGuid
'Get the GUID for all system HIDs.
'Returns: the GUID in HidGuid.
'The routine doesn't return a value in Result
'but the routine is declared as a function for consistency with the
other API calls.
'*******************************************************************

result = HidD_GetHidGuid(HidGuid)
Call DisplayResultOfAPICall("GetHidGuid")

'Display the GUID.
GUIDString = _
    Hex$(HidGuid.Data1) & "-" & _
    Hex$(HidGuid.Data2) & "-" & _
    Hex$(HidGuid.Data3) & "-"

For Count = 0 To 7
    'Ensure that each of the 8 bytes in the GUID displays two
     'characters.
    If HidGuid.Data4(Count) >= &H10 Then
        GUIDString = GUIDString & Hex$(HidGuid.Data4(Count)) & " "
    Else
        GUIDString = GUIDString & "0" & Hex$(HidGuid.Data4(Count)) & "
"
    End If
Next Count

ApdUSBPanel.lstUSB.AddItem "  GUID for system HIDs: " & GUIDString

'*******************************************************************
'SetupDiGetClassDevs
'Returns: a handle to a device information set for all installed
devices.
'Requires: the HidGuid returned in GetHidGuid.
'*******************************************************************

DeviceInfoSet = SetupDiGetClassDevs _
    (HidGuid, _
    vbNullString, _
    0, _
    (DIGCF_PRESENT Or DIGCF_DEVICEINTERFACE))
```

```
Call DisplayResultOfAPICall("SetupDiClassDevs")
DataString = GetDataString(DeviceInfoSet, 32)

'*******************************************************************
'SetupDiEnumDeviceInterfaces
'On return, MyDeviceInterfaceData contains the handle to a
'SP_DEVICE_INTERFACE_DATA structure for a detected device.
'Requires:
'the DeviceInfoSet returned in SetupDiGetClassDevs.
'the HidGuid returned in GetHidGuid.
'An index to specify a device.
'*******************************************************************

'Begin with 0 and increment until no more devices are detected.
MemberIndex = 0

Do
    'The cbSize element of the MyDeviceInterfaceData structure must be
     'set to
    'the structure's size in bytes. The size is 28 bytes.
    MyDeviceInterfaceData.cbSize = LenB(MyDeviceInterfaceData)
    result = SetupDiEnumDeviceInterfaces _
        (DeviceInfoSet, _
        0, _
        HidGuid, _
        MemberIndex, _
        MyDeviceInterfaceData)

    Call DisplayResultOfAPICall("SetupDiEnumDeviceInterfaces")
    If result = 0 Then LastDevice = True

    'If a device exists, display the information returned.
    If result <> 0 Then
        ApdUSBPanel.lstUSB.AddItem "  DeviceInfoSet for device #" &
CStr(MemberIndex) & ": "
        ApdUSBPanel.lstUSB.AddItem "  cbSize = " & CStr
(MyDeviceInterfaceData.cbSize)
        ApdUSBPanel.lstUSB.AddItem _
            "  InterfaceClassGuid.Data1 = " & Hex$
(MyDeviceInterfaceData.InterfaceClassGuid.Data1)
        ApdUSBPanel.lstUSB.AddItem _
            "  InterfaceClassGuid.Data2 = " & Hex$
(MyDeviceInterfaceData.InterfaceClassGuid.Data2)
        ApdUSBPanel.lstUSB.AddItem _
            "  InterfaceClassGuid.Data3 = " & Hex$
(MyDeviceInterfaceData.InterfaceClassGuid.Data3)
        ApdUSBPanel.lstUSB.AddItem _
            "  Flags = " & Hex$(MyDeviceInterfaceData.Flags)


        '***********************************************************
        'SetupDiGetDeviceInterfaceDetail
        'Returns: an SP_DEVICE_INTERFACE_DETAIL_DATA structure
        'containing information about a device.
        'To retrieve the information, call this function twice.
        'The first time returns the size of the structure in Needed.
        'The second time returns a pointer to the data in
           'DeviceInfoSet.
```

```
        'Requires:
        'A DeviceInfoSet returned by SetupDiGetClassDevs and
        'an SP_DEVICE_INTERFACE_DATA structure returned by
           'SetupDiEnumDeviceInterfaces.
        '*************************************************************

        'The final parameter passed in SetupDIGetDeviceInterfaceDetail
        'is an optional pointer to an SP_DEVINFO_DATA structure.
        'If retrieving the structure, set the size of
           'MyDeviceInfoData:
        'MyDeviceInfoData.cbSize = Len(MyDeviceInfoData)
        'and pass the structure's address (declare the parameter
           'ByRef)
        'This application doesn't retrieve or use the structure.

        MyDeviceInfoData.cbSize = Len(MyDeviceInfoData)
        result = SetupDiGetDeviceInterfaceDetail _
           (DeviceInfoSet, _
           MyDeviceInterfaceData, _
           0, _
           0, _
           Needed, _
           0)

        DetailData = Needed

        Call DisplayResultOfAPICall("SetupDiGetDeviceInterfaceDetail")
        ApdUSBPanel.lstUSB.AddItem "  (OK to say too small)"
        ApdUSBPanel.lstUSB.AddItem "  Required buffer size for the
data: " & Needed

        'Store the structure's size.
        MyDeviceInterfaceDetailData.cbSize = _
            Len(MyDeviceInterfaceDetailData)

        'Use a byte array to allocate memory for
        'the MyDeviceInterfaceDetailData structure
        ReDim DetailDataBuffer(Needed)
        'Store cbSize in the first four bytes of the array.
        Call RtlMoveMemory _
            (DetailDataBuffer(0), _
            MyDeviceInterfaceDetailData, _
            4)

        'Call SetupDiGetDeviceInterfaceDetail again.
        'This time, pass the address of the first element of
           'DetailDataBuffer
        'and the returned required buffer size in DetailData.
        result = SetupDiGetDeviceInterfaceDetail _
           (DeviceInfoSet, _
           MyDeviceInterfaceData, _
           VarPtr(DetailDataBuffer(0)), _
           DetailData, _
           Needed, _
           0)

        Call DisplayResultOfAPICall(" Result of second call: ")
        ApdUSBPanel.lstUSB.AddItem "
MyDeviceInterfaceDetailData.cbSize: " & _
```

```
            CStr(MyDeviceInterfaceDetailData.cbSize)

        'Convert the byte array to a string.
        DevicePathName = CStr(DetailDataBuffer())
        'Convert to Unicode.
        DevicePathName = StrConv(DevicePathName, vbUnicode)
        'Strip cbSize (4 characters) from the beginning.
        DevicePathName = Right$(DevicePathName, Len(DevicePathName) -
4)
        'Strip 2 trailing nulls from the end.
        DevicePathName = Left$(DevicePathName, Len(DevicePathName) - 2)
        ApdUSBPanel.lstUSB.AddItem "  Device pathname: " &
DevicePathName

        '**************************************************************
        'CreateFile
        'Returns: a handle that enables reading and writing to the
           'device.
        'Requires:
        'The DevicePathName returned by
           'SetupDiGetDeviceInterfaceDetail.
        '**************************************************************

        'Create an event object to signal completion of a read file
        EventObject = CreateEvent _
            (0&, _
            True, _
            False, _
            "")
                'First false says auto reset, second true says start
                'as signalled but will
            'be set to non-signalled when thread owns it - then when
                'thread releases, it is
            'set to signalled
            ' set to non-signalled first  'True, _
        Call DisplayResultOfAPICall("CreateEvent")

        'Fill overlapped structure
        HIDOverlapped.offset = 0
        HIDOverlapped.OffsetHigh = 0
        HIDOverlapped.hEvent = EventObject

        HID = CreateFile _
            (DevicePathName, _
            GENERIC_READ Or GENERIC_WRITE, _
            (FILE_SHARE_READ Or FILE_SHARE_WRITE), _
            0, _
            OPEN_EXISTING, _
            FILE_FLAG_OVERLAPPED, 0)
           'This used to be 0, 0) but was changed for overlapped call


        Call DisplayResultOfAPICall("CreateFile")
        ApdUSBPanel.lstUSB.AddItem "  Returned handle: " & Hex$(HID) &
"h"

        'Now we can find out if it's the device we're looking for.

        '**************************************************************
```

```
        'HidD_GetAttributes
        'Requests information from the device.
        'Requires: The handle returned by CreateFile.
        'Returns: a HIDD_ATTRIBUTES structure containing
        'the Vendor ID, Product ID, and Product Version Number.
        'Use this information to determine if the detected device
        'is the one we're looking for.
        '***************************************************************

        'Set the Size property to the number of bytes in the
           'structure.
        DeviceAttributes.Size = LenB(DeviceAttributes)
        result = HidD_GetAttributes _
            (HID, _
            DeviceAttributes)

        Call DisplayResultOfAPICall("HidD_GetAttributes")
        If result <> 0 Then
            ApdUSBPanel.lstUSB.AddItem "  HIDD_ATTRIBUTES structure
filled without error."
        Else
            ApdUSBPanel.lstUSB.AddItem "  Error in filling
HIDD_ATTRIBUTES structure."
        End If

        ApdUSBPanel.lstUSB.AddItem "  Structure size: " &
DeviceAttributes.Size
        ApdUSBPanel.lstUSB.AddItem "  Vendor ID: " & Hex$
(DeviceAttributes.VendorID)
        ApdUSBPanel.lstUSB.AddItem "  Product ID: " & Hex$
(DeviceAttributes.ProductID)
        ApdUSBPanel.lstUSB.AddItem "  Version Number: " & Hex$
(DeviceAttributes.VersionNumber)

        'Find out if the device matches the one we're looking for.
        If (DeviceAttributes.VendorID = MyVendorID) And _
            (DeviceAttributes.ProductID = MyProductID) Then
                ApdUSBPanel.lstUSB.AddItem "  My device detected"
                MyDeviceDetected = True
        Else
                MyDeviceDetected = False
                'If it's not the one we want, close its handle.
                result = CloseHandle _
                    (HID)
                DisplayResultOfAPICall ("CloseHandle")
        End If
End If
    'Keep looking until we find the device or there are no more
     'left to examine.
    MemberIndex = MemberIndex + 1
Loop Until (LastDevice = True) Or (MyDeviceDetected = True)

If MyDeviceDetected = True Then
    FindTheHid = True
    Call GetDeviceCapabilities
Else
    ApdUSBPanel.lstUSB.AddItem " Device not found."
    FindTheHid = False
End If
```

```
End Function


'*****************************************************************
'*****************************************************************
Private Function GetDataString _
    (Address As Long, _
    Bytes As Long) _
As String

'Retrieves a string of length Bytes from memory, beginning at Address.
'Adapted from Dan Appleman's "Win32 API Puzzle Book"

Dim offset As Integer
Dim result$
Dim ThisByte As Byte

For offset = 0 To Bytes - 1
    Call RtlMoveMemory(ByVal VarPtr(ThisByte), ByVal Address + offset,
1)
    If (ThisByte And &HF0) = 0 Then
        result$ = result$ & "0"
    End If
    result$ = result$ & Hex$(ThisByte) & " "
Next offset

GetDataString = result$

End Function



'*****************************************************************
'*****************************************************************
Private Function GetErrorString _
    (ByVal LastError As Long) _
As String

'Returns the error message for the last error.
'Adapted from Dan Appleman's "Win32 API Puzzle Book"

Dim Bytes As Long
Dim ErrorString As String
ErrorString = String$(129, 0)
Bytes = FormatMessage _
    (FORMAT_MESSAGE_FROM_SYSTEM, _
    0&, _
    LastError, _
    0, _
    ErrorString$, _
    128, _
    0)

'Subtract two characters from the message to strip the CR and LF.
If Bytes > 2 Then
    GetErrorString = Left$(ErrorString, Bytes - 2)
End If
```

```
End Function

'**************************************************************
'**************************************************************
Private Sub cmdReadSingleReport_Click()

' This simple sub just handles debug window button click event

    Dim result As Integer

    result = ReadSingleReport()
End Sub


'**************************************************************
'**************************************************************
Public Function ReadSingleReport(Optional blnShowReadBuffer As Boolean
= True) As Integer

' This sub attempts to read a single report from APD-SA USB

    Dim NumberOfBytesRead As Long
    Dim result As Integer
    Dim Count As Integer
    Dim result2 As Integer
    Dim blnHasCompleted As Boolean
    Dim blnresult3 As Boolean

    '** Commented out because assume already checked for attached and
     'appropriate device
    'and to allow other calling function to set read buffer size
    'If DeviceDetected = True Then
    '    'The ReadBuffer array begins at 0, so subtract 1 from the
     'number of bytes.
    '    If Capabilities.InputReportByteLength > 0 Then
    '        ReDim ReadBuffer(Capabilities.InputReportByteLength - 1)
    '    End If
    'End If

    blnHasCompleted = ResetEvent(EventObject)
    '**With ApdUSBPanel.lstUSB
    '    .AddItem "ResetEvent returned " & CStr(blnHasCompleted)
    '    .ListIndex = .ListCount - 1
    'End With

    result = ReadFile _
        (HID, _
        ReadBuffer(0), _
        CLng(Capabilities.InputReportByteLength), _
        NumberOfBytesRead, _
        HIDOverlapped)
           'Used to be just: 0) for non-overlapped read-file

    '**Call DisplayResultOfAPICall("ReadFile")


    'Wait for ReadFile to complete or a timeout whichever comes first
     'at spec'ed timeout interval
```

```
result2 = WaitForSingleObject _
    (EventObject, _
    1000) '1 second (# of ms)

'**Call DisplayResultOfAPICall("WaitForSingleObject")

blnresult3 = GetOverlappedResult _
    (HID, _
    HIDOverlapped, _
    NumberOfBytesRead, _
    False)
    'False = don't wait (sinze already have the
       'WaitForSingleObject function above
'**Call DisplayResultOfAPICall("GetOverlappedResult")

'This following function is not in kernel32
'blnHasCompleted = HasOverlappedIoCompleted _
'    (HIDOverlapped)

'With ApdUSBPanel.lstUSB
'    .AddItem "HasOverlappedIoCompleted = " & Cstr
 '(blnHasCompleted)
'    .ListIndex = .ListCount - 1
'End With

'Optional debugging information:
'Debug.Print "Readfile result= "; GetErrorString(Err.LastDllError)
'Debug.Print "handle= "; DeviceHandle
'Debug.Print "report length= "; ReportLength
'Debug.Print "bytes read= "; NumberOfBytesRead
'For Count = 0 To UBound(ReadBuffer)
'    Debug.Print ReadBuffer(Count)
'Next Count

'For Count = 1 To UBound(ReadBuffer)
'    ApdUSBPanel.lstUSB.AddItem " " & CStr(Count) & ": " & Hex$
 '(ReadBuffer(Count))
'Next Count

'This is for when it looks like it read ok: i.e.: "overlapped io
 'in progress" from ReadFile
'and "Completed OK" from WaitForEvent, but really the ReadFile
 'returned 0 bytes - probably
'because the USB sent a NACK to an IN request

'If result = 0 And NumberOfBytesRead = 0 Then
'If NumberOfBytesRead = 0 Then
'    ReadSingleReport = 0
'    ApdUSBPanel.lstUSB.AddItem "ReadFile returned 0 - abandoning"
'  ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
'    GoTo EndOfReadFile
'End If

If result2 = WAIT_TIMEOUT Or result2 = WAIT_ABANDONED Then
    ReadSingleReport = 0
'**ApdUSBPanel.lstUSB.AddItem "ReadFile Timed Out"
'**ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1
ElseIf result2 = WAIT_OBJECT_0 Then
    ReadSingleReport = NumberOfBytesRead '=True, = 1 etc.
```

```
        If blnShowReadBuffer = True Then
            Call ShowReadBuffer
        End If

        '**With ApdUSBPanel.lstUSB
        '    .AddItem "Bytes read in ReadSingleReport: " & Cstr
           '(ReadSingleReport) 'should show number of bytes read
        '    .ListIndex = .ListCount - 1
        'End With
    End If

EndOfReadFile:

End Function



'******************************************************************
'******************************************************************
Private Sub DisplayResultOfAPICall(FunctionName As String)

'This sub displays the results of an API call.

Dim ErrorString As String

ApdUSBPanel.lstUSB.AddItem ""

ErrorString = GetErrorString(err.LastDllError)

ApdUSBPanel.lstUSB.AddItem FunctionName
ApdUSBPanel.lstUSB.AddItem "  Result = " & ErrorString

'Scroll to the bottom of the list box.
ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

End Sub



'******************************************************************
'******************************************************************
Private Sub Form_Unload(Cancel As Integer)

' This sub simply ensures that if, on closing the application,
' there is still USB handle open, that it is closed.

     Call ShutdownUSB
End Sub

'******************************************************************
'******************************************************************
Private Sub GetDeviceCapabilities()

' This sub uses USB HID driver APD to get information
' about the attached USB HID device.

'*******************************************************************
'HidD_GetPreparsedData
'Returns: a pointer to a buffer containing information about the
```

```
'device's capabilities.
'Requires: A handle returned by CreateFile.
'There's no need to access the buffer directly,
'but HidP_GetCaps and other API functions require a pointer to the
'buffer.
'*******************************************************************

Dim ppData(29) As Byte
Dim ppDataString As Variant

'Preparsed Data is a pointer to a routine-allocated buffer.
result = HidD_GetPreparsedData _
    (HID, _
    PreparsedData)
Call DisplayResultOfAPICall("HidD_GetPreparsedData")

'Copy the data at PreparsedData into a byte array.
result = RtlMoveMemory _
    (ppData(0), _
    PreparsedData, _
    30)
Call DisplayResultOfAPICall("RtlMoveMemory")

ppDataString = ppData()
'Convert the data to Unicode.
ppDataString = StrConv(ppDataString, vbUnicode)

'*******************************************************************
'HidP_GetCaps
'Find out the device's capabilities.
'For standard devices such as joysticks, you can find out the specific
'capabilities of the device.
'For a custom device, the software will probably know what the device
'is capable of,
'so this call only verifies the information.
'Requires: The pointer to a buffer containing the information.
'The pointer is returned by HidD_GetPreparsedData.
'Returns: a Capabilites structure containing the information.
'*******************************************************************
result = HidP_GetCaps _
    (PreparsedData, _
    Capabilities)

Call DisplayResultOfAPICall("HidP_GetCaps")
ApdUSBPanel.lstUSB.AddItem "  Last error: " & ErrorString
ApdUSBPanel.lstUSB.AddItem "  Usage: " & Hex$(Capabilities.Usage)
ApdUSBPanel.lstUSB.AddItem "  Usage Page: " & Hex$
(Capabilities.UsagePage)
ApdUSBPanel.lstUSB.AddItem "  Input Report Byte Length: " &
Capabilities.InputReportByteLength
ApdUSBPanel.lstUSB.AddItem "  Output Report Byte Length: " &
Capabilities.OutputReportByteLength
ApdUSBPanel.lstUSB.AddItem "  Feature Report Byte Length: " &
Capabilities.FeatureReportByteLength
ApdUSBPanel.lstUSB.AddItem "  Number of Link Collection Nodes: " &
Capabilities.NumberLinkCollectionNodes
ApdUSBPanel.lstUSB.AddItem "  Number of Input Button Caps: " &
Capabilities.NumberInputButtonCaps
```

```
ApdUSBPanel.lstUSB.AddItem "  Number of Input Value Caps: " &
Capabilities.NumberInputValueCaps
ApdUSBPanel.lstUSB.AddItem "  Number of Input Data Indices: " &
Capabilities.NumberInputDataIndices
ApdUSBPanel.lstUSB.AddItem "  Number of Output Button Caps: " &
Capabilities.NumberOutputButtonCaps
ApdUSBPanel.lstUSB.AddItem "  Number of Output Value Caps: " &
Capabilities.NumberOutputValueCaps
ApdUSBPanel.lstUSB.AddItem "  Number of Output Data Indices: " &
Capabilities.NumberOutputDataIndices
ApdUSBPanel.lstUSB.AddItem "  Number of Feature Button Caps: " &
Capabilities.NumberFeatureButtonCaps
ApdUSBPanel.lstUSB.AddItem "  Number of Feature Value Caps: " &
Capabilities.NumberFeatureValueCaps
ApdUSBPanel.lstUSB.AddItem "  Number of Feature Data Indices: " &
Capabilities.NumberFeatureDataIndices

'*******************************************************************
'HidP_GetValueCaps
'Returns a buffer containing an array of HidP_ValueCaps structures.
'Each structure defines the capabilities of one value.
'This application doesn't use this data.
'*******************************************************************

'This is a guess. The byte array holds the structures.
Dim ValueCaps(1023) As Byte

result = HidP_GetValueCaps _
    (HidP_Input, _
    ValueCaps(0), _
    Capabilities.NumberInputValueCaps, _
    PreparsedData)

Call DisplayResultOfAPICall("HidP_GetValueCaps")

'ApdUSBPanel.lstUSB.AddItem "ValueCaps= " & GetDataString((VarPtr
(ValueCaps(0))), 180)
'To use this data, copy the byte array into an array of structures.

End Sub




'*****************************************************************
'*****************************************************************
Private Sub SendReportToTheHID()

'Send a report to the APD-SA USB device.

Dim Count As Integer
Dim NumberOfBytesRead As Long
Dim NumberOfBytesToSend As Long
Dim NumberOfBytesWritten As Long
Dim WriteSuccess As Boolean

'In most cases, once a device has been detected it will remain
'available.
'But it's possible that the device has been removed, or removed and
```

```
'reattached
'(with a new handle). We need to handle each of these cases.

If DeviceDetected = True Then
    WriteSuccess = DoAWriteFile
    'If the write attempt failed, it may be because the device was
     'removed,
    'then reattached. So try to find it again.
    If WriteSuccess = False Then
        DeviceDetected = FindTheHid
        'If success, try to write to the device.
        If DeviceDetected = True Then
            WriteSuccess = DoAWriteFile
        End If
    End If
Else
    'If the device isn't detected, try to find it.
    DeviceDetected = FindTheHid
    'If success, try to write to it.
    If DeviceDetected = True Then
        WriteSuccess = DoAWriteFile
    End If
End If

If WriteSuccess = True Then
    ApdUSBPanel.lstUSB.AddItem " OutputReportByteLength = " &
Capabilities.OutputReportByteLength
    ApdUSBPanel.lstUSB.AddItem " NumberOfBytesWritten = " &
NumberOfBytesWritten
    ApdUSBPanel.lstUSB.AddItem " Report ID: " & SendBuffer(0)
    ApdUSBPanel.lstUSB.AddItem " Output Report Data:"
    For Count = 1 To UBound(SendBuffer)
        ApdUSBPanel.lstUSB.AddItem " " & Hex$(SendBuffer(Count))
    Next Count
Else
    ApdUSBPanel.lstUSB.AddItem "Unable to write to device."

End If
'Scroll to the bottom of the list box.
ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

End Sub


'*******************************************************************
'*******************************************************************
Public Sub ShutdownUSB()

' This sub performs tasks that must execute when the program ends
' or when USB session has ended and in order to close the USB
' connection.
'
' Send a report to the device to tell the device to
' tell the server to stop reading from the device.
' Call StopReadingReports

On Error Resume Next

'Close the open handle to the device.
```

```
result = CloseHandle _
    (HID)
Call DisplayResultOfAPICall("CloseHandle (HID)")

'Free memory used by SetupDiGetClassDevs
'Nonzero = success
result = SetupDiDestroyDeviceInfoList _
    (DeviceInfoSet)
Call DisplayResultOfAPICall("DestroyDeviceInfoList")

result = HidD_FreePreparsedData _
    (PreparsedData)
Call DisplayResultOfAPICall("HidD_FreePreparsedData")

ApdPanel.UseStandAloneAPD.Checked = False
ApdPanel.UseDataAcquisitionCardAPD.Checked = True

ApdUsb.ToggleApdUsbOptions (False)

'Call ApdPanel.UseStandAloneAPD_Click

MsgBox "Closed USB connection..."

End Sub

'*******************************************************************
'*******************************************************************
Private Sub StopReadingReports()

'Send an Output report to the device with byte 0 = FFh.
'This tells the device to notify the server that it should stop
'reading from the device.
'When the device reads the report, it sets byte 0 of its next Input
'report to FFh.
'When the server reads this report, it knows that it should stop
'attempting to
'read Input reports from the device. This enables the server to close.

     OutputReportData(0) = &HFF
     Call SendReportToTheHID

End Sub

'*******************************************************************
'*******************************************************************
Private Sub Download()

' This sub downloads the compiled EZ-USB firmware to the EZ-USB
' device in the APD-SA USB

'Private Sub cmdDownloadFile_Click()
Dim dirname As String
Dim x, lastx As Integer

'  CommonDialog1.InitDir = GetSetting("EZ-USB_DownloadHex",
'"Defaults", "Directory", "")
'
'    CommonDialog1.ShowOpen
'    If CommonDialog1.FileName <> "" Then
```

```
'        x = -1
'        lastx = 0
'        Do While x <> 0
'            x = InStr(lastx + 1, CommonDialog1.FileName, "\")
'            If x <> 0 Then: lastx = x
'        Loop
'
'        dirname = Left(CommonDialog1.FileName, lastx)
'
'        SaveSetting "EZ-USB_DownloadHex", "Defaults", "Directory",
'dirname
'
'        gDriver = cmbDriverName.Text
'
'        LoadHexFile CommonDialog1.FileName
'    End If

ApdUSBPanel.lstUSB.AddItem "Downloading firmware (attempting...)"
ApdUSBPanel.lstUSB.ListIndex = ApdUSBPanel.lstUSB.ListCount - 1

Call Memory.Main
Call InitDownload

'dirname =
'"C:\Windows\Desktop\OldDesktop\MastersThesis\Software\Apd200\"

'This is from host desktop program - 1/04
'dirname = "C:\Cypress\USB\Examples\EzUsb\HID_Mouse\EZ-USB
'Mouse\ezmouse\"

'This is for laptop and future versions
dirname = "C:\Apd\Firmware\"

'SaveSetting "EZ-USB_DownloadHex", "Defaults", "Directory", dirname

gDriver = "Ezusb-0"

LoadHexFile dirname & "ezmouse.hex"

End Sub


'*******************************************************************
'*******************************************************************
Private Sub InitDownload()
'Private Sub Form_Load()

' This sub initiates the firmware download to the EZUSB on APD-SA USB

Dim index As Integer
Dim sDriverName As String
Dim hDriver As Long
Dim DeviceDescriptor As USB_DD


   ' find all the attached EZ-USB devices
   ' add device names to combo-box list
    For index = 0 To MAX_USB_DEV_NUMBER - 1
        sDriverName = "Ezusb-" & index
```

```
        hDriver = OpenDriver(sDriverName)
        If hDriver > 0 Then
            'cmbDriverName.AddItem sDriverName
            CloseHandle hDriver
        End If
    Next

    'ProgressBar1.value = 0

    'If cmbDriverName.ListCount > 0 Then
    '    cmbDriverName.Text = cmbDriverName.List(0)
    '    GetDeviceDescriptor cmbDriverName.Text, DeviceDescriptor
    'Else
    '    ErrMsg (eBadDriver)
    '    End
    'End If


End Sub



'*******************************************************************
'*******************************************************************
Private Sub DoneDownload()

' This sub clears memory when download has finished.

Dim Segment As CSegment
' release memory

For Each Segment In gFirmwareCache
    Set Segment = Nothing
Next

Set gFirmwareCache = Nothing

End Sub
```

APPENDIX G: SUBJECT DATA

Table G.1. Random numbers for measurement order; a one indicates that APD100 was used first

1	2	2	2	2	1	2	1	2	2	1	1	2

Table G.2. Subject Data: age, height, weight, self-reported respiratory issues

Subject	Age (Yrs)	Sex	Height (m)	Mass (kg)	Respiratory Issues
1	26	M	5'10”	165	No
2	60	F	5'2”	120	No
3	58	M	5'10”	168	No
4	26	F	5'5”	123	No
5	21	M	6'2”	155	No
6	19	F	5'7”	125	No
7	44	M	5'8”	207	Smoked heavily since 18
8	33	M	5'10”	230	No
9	28	M	6'	200	Smoker
10	18	F	5'2”	125	Mild asthma – inactive
11	23	F	5'2”	122	Mild asthma – inactive
12	21	F	5'5”	120	No
13	33	F	5'9”	135	No

Table G.3. Subject Data for APD100 measurements for 13 subjects

Subject	Average RR (cmH2O/Lps)	Inh RR (cmH2O/Lps)	Exh RR (cmH2O/Lps)	Avg Flow (Lps)	Inh Flow (Lps)	Exh Flow (Lps)	Avg % Pert	Inh % Pert	Exh % Pert	Inh Perts	Exh Perts	Frequency (Hz)
1	2.03	1.87	2.19	1.20	0.91	1.48	32.4	40.7	24.2	127	243	7.6
2	3.72	3.25	4.18	0.83	0.86	0.80	37.6	51.4	23.9	120	113	6.3
3	2.91	2.68	3.13	0.56	0.69	0.44	27.4	36.3	18.6	120	114	6.6
4	3.31	2.92	3.71	1.08	1.16	1.00	25.6	34.8	16.3	120	112	7.1
5	2.20	1.84	2.57	1.73	1.76	1.70	25.8	29.3	22.3	120	113	6.6
6	3.19	2.73	3.65	0.67	0.85	0.50	39.6	54.8	24.4	114	120	6.3
7	4.42	3.83	5.01	0.59	0.66	0.51	27.6	39.1	16.1	120	119	6.7
8	2.72	2.45	2.98	0.83	0.80	0.85	31.6	40.2	22.9	120	116	7.1
9	3.09	3.21	2.97	0.90	1.15	0.64	50.6	70.0	31.2	116	120	6.6
10	3.39	3.36	3.42	0.67	0.77	0.56	30.6	38.1	23.1	114	120	6.8
11	3.08	2.63	3.53	0.49	0.52	0.45	27.4	35.2	19.7	120	120	6.8
12	3.77	3.47	4.08	0.97	0.89	1.05	17.4	19.6	15.1	120	105	7.1
13	2.79	2.38	3.19	0.60	0.56	0.63	18.4	20.1	16.7	120	116	7.6

Table G.4. Subject Data for APD-SA measurements for 13 subjects

Subject	Average RR (cmH2O/Lps)	Inh RR (cmH2O/Lps)	Exh RR (cmH2O/Lps)	Avg Flow (Lps)	Inh Flow (Lps)	Exh Flow (Lps)	Avg % Pert	Inh % Pert	Exh % Pert	Inh Perts	Exh Perts	Frequency (Hz)
1	2.02	1.80	2.24	1.20	1.13	1.28	25.4	30.5	20.4	177	287	7.3
2	4.15	3.48	4.81	0.69	0.70	0.69	28.4	38.1	18.7	138	131	6.6
3	3.06	2.93	3.18	0.78	0.93	0.62	27.1	36.4	17.7	135	182	7.1
4	3.50	3.16	3.84	1.03	1.18	0.88	21.6	29.8	13.5	135	133	6.0
5	2.23	1.87	2.58	1.70	1.89	1.52	17.0	22.2	11.7	134	130	7.1
6	3.16	2.96	3.36	0.78	0.85	0.70	25.7	34.4	16.9	155	135	7.5
7	4.65	4.21	5.08	0.52	0.49	0.55	26.3	33.8	18.8	165	137	8.0
8	2.76	2.41	3.11	0.49	0.51	0.48	22.6	28.2	17.0	148	90	7.1
9	3.14	3.08	3.20	0.48	0.54	0.41	29.4	39.0	19.9	161	132	7.9
10	3.91	3.59	4.24	0.78	0.79	0.77	22.0	28.7	15.2	163	131	6.9
11	3.15	2.69	3.62	0.44	0.50	0.39	30.5	40.7	20.3	221	134	7.4
12	3.78	3.53	4.04	0.94	0.97	0.91	12.2	14.1	10.2	130	140	8.0
13	2.97	2.51	3.42	0.51	0.52	0.51	13.4	15.1	11.7	156	111	8.3

APPENDIX H: EQUIPMENT AVAILABLE FOR APD-SA DESIGN

Electronics that can be implemented in a hand held or at least compact arrangement will be considered here since the alternative home-use RMD, the APD, is the research focus. The APD requires data collection via ADC, data analysis and interface capabilities: display, buttons and possibly input-output. To perform data analysis, some sort of digital computing IC must be incorporated. Examples include: smaller microcontrollers (MCUs), generally targeted for less complex calculation or system control – a complex system might be built around several linked MCUs; CPUs, generally able to perform more complex calculations and system control, and designed for faster operation speed – a complex system might be entirely controlled by a single CPU; and DSPs, ICs somewhere between MCUs and CPUs that are optimized for complex calculations, especially those frequently required in digital filters or similar signal manipulation in which members of a set of data are continually multiplied and added – a complex system dedicated to signal processing, such as a multimedia device, might be designed around a DSP.

Memory, Storage and IO

To perform a data analysis task, an IC must be able to store and move data as well as permanently retain the instructions it is required to execute in order to process the data. Many MCUs and DSPs contain enough temporary data storage space and

RAM onboard to perform their designated tasks, while CPUs typically have enough RAM onboard to prepare for the next few routines, but not their entire set of designated tasks. It is necessary to add peripheral RAM chips to augment the chip's inherent memory, unless the data space and space to store the program code on the IC are particularly small. Generally, RAM is intended only for temporary data storage and does not retain data after power has been removed from the chip, though other configurations allow long-term RAM data storage.

To operate a data analysis system, it necessary to permanently or at least persistently store data until it requires alteration. Stored information that can be altered is often called firmware because it is more permanent than software but not irreversibly hardwired like hardware. Program code, the set of instructions for the IC, will need to be stored indefinitely or until the program needs to be updated. This type of memory is usually intended only to be read from, not written to, and is thus called read only memory (ROM). If a program is finalized and slated for mass production, then the code can be built into each new chip in a custom order from the IC manufacturer. Alternatively, one-time programmable chips can be obtained and programmed in the development facilities with an ultraviolet (UV) programmer.

If prototype code is used, then it is desirable to be able to reprogram the ICs, and thus reprogrammable ICs would be purchased. The code is similarly burned into the chip's memory itself via UV or into a memory chip which is then accessed when power is first applied to the circuit. Reprogrammable chips are often electrically erasable programmable read-only memory (EEPROM) chips, whether the memory sits inside an MCU or it resides in an external memory IC. These chips are programmed with a series

of commands that activate an elevated voltage that allows data to be written to the chip. This is performed in an EEPROM programming machine or, in many ICs, can be achieved by sending a command to the chip, for example through its serial port controller, that will cause it to program its own memory by generating a higher voltage required for programming. FLASH memory is a form of easily alterable EEPROM and as such retains data after power has been removed. An alternative method to store data is to have a battery apply small amounts of current to a static RAM (SRAM) chip – just enough to allow it to retain its data. This configuration retains the advantage of high speed data movement to and from the SRAM with the advantage of data persistence, or long-term storage. Other forms of storage can be used to store program code and large amounts of data, approaching the qualities of hard disks used in desktop computers. These include FLASH variants like CompactFlash and IBM Microdrives, that retain their data without power, but are a bit slower to access.

Some ICs incorporate controls on board for moving data in and out of the chip or circuit board while others require an external IO controller. Most chips incorporate some method for data transfer with a neighboring chip, often serial protocols. These protocols include Phillips IC-IC (I2C), serial peripheral interface (SPI) and various other serial inter-chip protocols, some particular to a manufacturer, based on one to four or more wired connections between chips on which data is transmitted one bit at a time. Parallel data transmission is also used in which nibbles (4 bits), bytes (8 bits) or words (16 to 32 to 64 or even 128 bits on internal data buses in DSPs) are transferred at once between chips, or computers in the standard LPT parallel port case.

It is often necessary to move data from one circuit, or hardware platform, to another. A serial protocol such as RS232 or RS485 is usually reserved for long distance transfer to another computer and uses digital pulse voltages in conjunction with optional error checking routines and data transmission configurations that attempt to preserve the integrity of the data pulses in a noisy environment.

Some ICs either include the capability to control, or have the sole responsibility to control, infrared (IrDA) data transmission, universal serial bus (USB) or Ethernet data transmission. IrDA is typically used for short distance peripheral to computing device data communication. USB is typically used for desktop computer to peripheral communication in which up to 128 devices can be connected to a computer via hubs, with each cable link up to 5 meters in length. Ethernet is designed for much longer distances between connections, and includes provisions in its protocol for networks of many computers. Still other protocols such as IEEE-1394, also called firewire, allows inter-peripheral communication between multimedia devices and a computer. In the cases of chip-to-chip communication, I2C and SPI are standardized and fairly simple since they are low level communications protocols, implemented during the circuit board and firmware design process. Protocols designed for communication from one circuit board to another, such as RS232, RS485, USB, and Ethernet, have more complex protocols standardized by a governing agency (*e.g.* www.usb.org, USB implementers forum). These more complex communication methods are supported by numerous texts written to extract the essential elements from the published standards, such as Jan Axelson's books like Parallel Port Complete (1997), Serial Port Complete (1998) and USB Complete (2001). These protocols are more complex because they are designed

for entire computing systems to communicate and must be general enough to be independent of the type of hardware used. The protocols must also allow harmonious communication when multiple communication activities simultaneously occur.

Hardware and Software Integration Options

A successful design should include hardware that sufficiently supports the software needs dictated by the data manipulation scheme devised for the application, in this case the stand-alone APD. Many combinations of ICs are possible from highly integrated MCUs to CPUs and peripherals. Designing, prototyping and assembling circuits brings challenges in signal routing to ensure data integrity and the costs in the tools required to create the designs. Custom outsourced circuit design solves some of these problem but introduces costs for the service. Commercially available hardware platforms may be used. These offer the cost benefits of mass production but frequently offer capabilities beyond those needed for the application and their associated cost. Some commercially mass produced platforms fit a standardized specification, such as PC104 and thus, upon purchase, the user can be assured of particular hardware capabilities. A balance among these options must be achieved with the APD's purpose in mind. Each hardware configuration determines the type of software that may be used to implement the data analysis algorithms.

Platform Software

Software Options. Any platform for the device will incorporate an MCU or CPU for its processing and thus presents the need to address programming methods. The options range from direct assembler programming to higher level programming in an integrated development environment (IDE). To use assembler, the code would be compiled and programmed into the MCU or its memory peripherals via debugging port on a circuit or an EEPROM burner. This approach is typical for smaller MCUs and demands that the code be specific to the selected MCU model. Another option for both MCUs and more complex microprocessors approaching PC CPUs, is to use an IDE in which higher-level code, such as American National Standards Institute (ANSI)-compliant C, is compiled upon completion and loaded onto the MCU or CPU. The benefits of assembler programming for smaller MCUs is the simplicity and overall low-cost. The disadvantage of this method is the specificity of the code and the resulting lack of portability across platforms should a new MCU or CPU be selected. Further, the specificity makes it less accessible to other programmers should quick modifications be required. The advantage of using an IDE for simpler MCUs is that the original code becomes portable, if written in a commonly used language, such as C, and the code is more readily accessible. The disadvantage of this method is that the IDE may be more costly and will still require tools for loading the compiled code onto the platform. If an IDE is used that is intended for more complex CPUs used in typical PCs, the IDE may be costly, but the programming methods are potentially well-known by programmers, and the methods of loading the code onto a platform more closely resemble those used

for PC data transfer. This simplifies the debugging process and the tools required to prototype and produce a system.

BRE-UMCP Software Resources. The ENBE Department currently has licenses for Microsoft Visual Studio which includes Visual C++ and Visual Basic IDEs. Writing original code in ANSI C will facilitate any future code modifications and ensure the portability of the code, as long as a C-compiler is available for the selected platform (MCU or CPU). Visual C++ is particularly useful for platform design because it is able to time profile its code, meaning it is possible to determine how long a portion of code takes to execute. Thus, the code for time-critical tasks can be optimized to execute as quickly as possible. If the compiled code can be used directly from the Visual Studio IDE, than the costs of a C-compiler and IDE are eliminated from the project, while the portability and accessibility of the code are maintained. A potential pitfall arises in this case however, in which the platform that is able to run this code too closely resembles a PC and hence integrates many unused functions that inflate the cost of each APD-SA. Clearly, a balance among these issues must be sought.

Platform Hardware

BRE-UMCP Resources: Wire-wrapped and Low-cost PCB Fabrication. Electronics manufacturing capabilities within the department are limited to wire-wrapping or the acquisition and use of low-cost PCB fabrication systems such as print and transfer copper etching systems. Preliminary tests showed wire-wrapped prototype

circuits to be viable up to about 8 or 16MHz. At higher frequencies, stray capacitances and wire inductance caused considerable ringing or degraded digital pulse shapes. These 8 and 16 MHz constraints might allow for wire-wrapped APD circuits using typical 8-bit 8 MHz microcontrollers (MCUs) such as those from Motorola or PIC and numerous other manufacturers. Using such a system however would require the use of several peripheral chips for data memory and IO management since these chips are limited in their IO capabilities and storage. With an increase in chip count, the time required to manufacture one board would become very large. Recently, a wire-wrapped board with approximately 300 connections required 40 to 80 hours to complete. Furthermore, the less constrained signal lines in complex wire-wrapped designs increased the likelihood of signal crosstalk and errors in the wrapping process.

PCB fabrication using etching kits poses similar signal and fabrication complexity limitations. Typical etching kits are limited in their trace resolution and allow one double-sided layer. With a single layer, the routing of complex tracings for a microcontroller poses a challenge. Precise etches can be made with laser printed tracings. However, they are of limited resolution due to the printing process. Complex, fine-trace circuits also require precise drilling for component holes. A successful PCB fabrication would require proper layout derived from the appropriate layout software. ExpressPCB offers custom layout software for free, but uses the software in its in-house fabrication. OrCad or other CAD programs for electronics are viable options but are costly. Free versions of OrCad were found to be inadequate for the project due to their lack of components, since OrCAD has discontinued its practice of providing IC models for free. In any case, the process of developing a successful PCB requires attention to

signal routing for proper current carrying capacities, minimized trace length and clean signals. Such details require a large time commitment and possibly several PCB prototype revisions, potentially a process better suited for future platform development once the success of the data processing methods has been proven.

Contracted Custom PCB Fabrication. Using a commercial PCB fabricator shifts outside of the department the cost of tooling to produce the actual PCB. This allows for more precise and complex PCB designs that would facilitate the use of large chip-count designs in small PCB dimensions fabricated from more expensive automated machinery. An extensive list of PCB fabricators can be found through Thomas Register and through an internet search engine, Google's, online PCB service directory at http://directory.google.com/Top/Business/Electronics_and_Electrical/Contract_Manufacturers/Printed_Circuit_Boards/Fabrication/. Firms such as ExpressPCB (www.expresspcb.com) and Circuit Express, Inc. (www.circuitexpressinc.com) offer short-lead time PCB manufacture for less than $100 in minimum quantities of two to six double-layer 20-square inch boards. Other companies offer more precise PCB manufacturing options at a higher price: Hughes Circuits, Inc. offers 4-6 layers boards at a $1100 typical minimum per lot (Glatts, 2002); Douglas Electronics offers double-layer boards at $350 per lot for 0.062-in min trace at 8-mil thickness, 16-sq. in, a configuration in 4 layers is closer to $1050 (Vierra, 2002).

Contracting a PCB manufacturer would still pose challenges on the remaining PCB design and APD-SA manufacture. Outsourcing the PCB manufacture would require custom circuit layout and with it the associated layout tools. The PCB required for an APD control and analysis platform demands high-speed digital traces and cross-talk protected analog traces and thus would require some time devoted to adjusting the layout to ensure signal integrity, ideally through simulation or prototype revision. Manufacturing each subsequent APD-SA would be simplified in the PCB fabrication step, but would still require component placement. Soldering the components would require further chip placement costs either to acquire the proper tools or simply in the required time. Any high-density pin chip designs such as small outline (SO) would greatly reduce PCB size and cost, but potentially require further specialized chip placement tools for manufacture. Thus, the required cost and time make inefficient the use of a custom prototype printed circuit board (PCB) for the entire circuit at the present stage of APD development in which the focus is to design and test data processing architecture in a stand-alone, fast-response configuration.

Wire-wrap and PCB fabrication pose particular challenges. Wire-wrapping poses signal limitations and thus restricts MCU and system architecture towards larger chip count circuits that require longer manufacture times and are less accessible for modification. PCB fabrication would require the acquisition of appropriate layout software and etching materials. Though some of these are available in limited and low-cost versions, the design task for the APD-SA would incur a large commitment to

layout and PCB development, even if outside manufacturers are contracted to produce the actual PCB.

Custom PCB Development Requirements for MCU Programming. If a custom PCB were implemented, the investment in MCU programming and debugging must be considered. Developing complex designs that incorporate faster MCUs requires expensive apparatus approaching the cost of personal computer (PC) motherboard design since the instruction sets and programming interfaces become more complex. Thus, custom APD-SA circuit design would most likely be limited to more compact MCU designs. Figure H.1 typifies a hardware design for a system using MCUs for peripheral data control built around a DSP core as an example of the types of considerations required in a custom hardware design. These hold the advantage of simplified instruction sets and simpler system designs, allowing a system to built with bare-bones programming. Such programming however would be specific to each chip manufacturer and model. For example, the instruction set and chip peripherals would be specifically programmed for each manufacturer, *e.g.* Motorola, Atmel, PIC, Analog Devices, Zilog and others. This requires an investment in the appropriate compilers, though in some cases, assemblers and limited functionality compilers are available for free or at low cost. Motorola and many others provide free assembler language compilers. Free general Unix-based public license (GNU) compilers are available for some Motorola chips without much support. MicroChip, Inc. provides a free entry level Windows-based development environment for some of its smaller MCUs. Several

third-party developers offer ANSI C compliant compiler development environments for many MCU manufacturers. In each of these cases as the MCU processing power increases, the development platform costs increase. Typical Motorola or Intel development environments for powerful DSP or 16-bit or high-speed 8-bit MCUs can run into the $1000 range or more. In any case, whether a free compiler or low-cost development environment is used for prototyping, the software must be downloaded onto a chip. Thus some portion of the design time will be required simply for MCU programming from a host computer, for example via serial port, not just for prototype functionality in its eventual application. Alternatively, a chip programmer might be purchased for those MCUs that are available in erasable electrically programmable read only memory (EEPROM) dual inline pin (DIP) programmable packages. Each programming machine is capable of programming only chips of an architecture for which it was designed, *e.g.* DIP or peripherally arrange pins for insertion into a chip carrier (PLCC). In any case, once an MCU was selected, it would determine the investment in corresponding programming tools - software and hardware – thus making costly future changes in platform architecture.

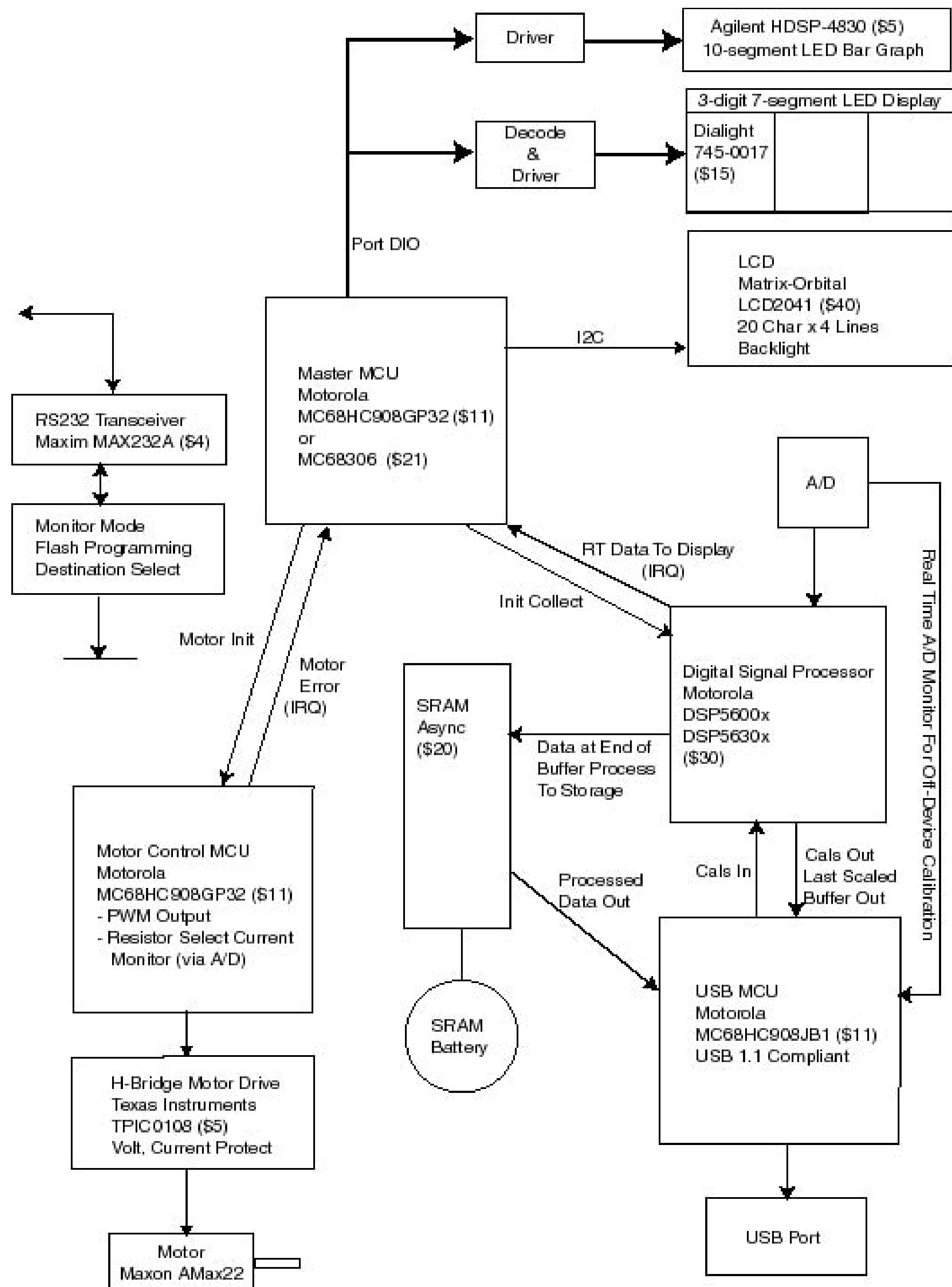


Figure H.1. Example of an MCU/ DSP custom stand-alone APD hardware design

It thus appears that any custom PCB development requires an investment in proper tools. The time to learn and assemble the prototyping and programming system

are considerable. Future modifications would require that the person modifying the design become acquainted with a set of tools specific to one MCU architecture – a valuable experience – but one that would preclude quick modifications, adjustments or gaining a more universally applied knowledge.

It seems the more immediately important focus now is on the successful implementation of data processing algorithms in order to answer questions such as: Will the system work? How should the data analysis parameters be adjusted to provide a fast-responding sensitive stand-alone device? In the future, focus might then be shifted towards reducing costs through custom PCB manufacture and component selection.

Pre-Fabricated Embedded Platform. A pre-fabricated platform speeds much of the prototyping process. In implementing a pre-fabricated platform such as a single-board computer, the manufacturing company assumes the initial capital investment of optimizing signal layout, PCB fabrication, prototyping and designing a programming interface and environment, thus leaving the user to benefit from a tested hardware platform that can be programmed immediately. The cost of a platform ranges from near $200 to $1000 and demonstrates the benefit of mass produced platforms from an external source (see table H.1). This prototyping method allows for immediate focus on the data processing and system control architecture. The more closely a platform resembles a PC, the more likely it is to be programmable through more readily accessible means. For example, many industry standard architecture single-board computers, such as those meeting the PC104 specification, can be loaded with PC

operating systems and run PC-developed programs. Thus additional software need not be purchased, and the programs can be loaded onto the platforms using standard serial interfaces. Most of these products are designed to be industrially embedded and provide maximum returns for the capital investment and so are generally supported by the manufacturer for years to come. Zworld, Inc. one embedded hardware manufacturer, has never discontinued or changed a product still used by a customer (Wills, 2001). VersaLogic, Inc. (2002) states that it uses chips available from multiple vendors to ensure product longevity. Many platform manufacturers have stated similar support and availability time windows, citing their large initial investment in a stable industry-accepted platform as a reason to support and manufacture their products for long periods of time, when compared with the rate at which personal computer platforms change (Wills, 2001).

Table H.1. Typical commercially manufactured embedded hardware platforms

Manufacturer	Model	Features	Disadvantage	Pricing
Blue Chip Technology	MICRON PC104 SBC	200MHz, 64MB RAM, CRT, EN, USB, LPT, SP, RTC, MSE, KBD, FDD, HDD	Excessive features, large power consumption	$850.00 (ea) (£545.00) (Keith Heaviside, 7.18.02)
Zworld	Omega, Wildcat, Smartcat SBC and Dynamic C 32	ADC, SP, EN, SP, RTC, LCD expansion, DIO, LP	No USB, slow – for lower-sampling rate control applications	$400 (kit) $200 – 300 subsequent
Versa Logic	EPM-CPU-3 (PC104)	AMD SC520 133MHz, EN, SP, FDD, HDD	Excessive features, no USB	$375- $400

WinSystems	PPM-520 (PC104) PPM-TX is PPM-520 with USB and MMX P166 or P233 MHz	AMD Elan SC520 133 MHz, +5, LPT, SP, EN, KBD, MSE, HDD, FDD	Excessive features, no USB on 520	$600 to $1000 depending on features.
MicroComputer Systems, Inc.	PC104 x386 MSI-CM387	Low cost PC104 SBC, FDD, EN, +5, SP, LPT, FDD, CRT	No USB, some excessive port components, slow PCU	$220 (ea)
Intrinsyc, Inc.	CerfBoard	Intel StrongARM 133/206 MHz, <PC104 size, USB, Win CE 3.0, SP, EN, 16 MB FLASH, 32 MB SDRAM standard	Some excessive components, development platform req’d	$1000 development kit; $350 subsequent, small quantity. (Kaarto, 2002)
AmPro, Inc.	CoreModule P5e (PC104)	SP, IrDA, USB, LPT, KBD, MSE, FDD, +5, P266 MHz, CRT, HDD	High power consumption, 6 to 8 watts, custom memory not field upgradable, excessive features	
Arius, Inc.	PC104C31	TI DSP PC104, TMS320C31 80 MHz, SP, AIO option	Expensive TI compiler required; no USB	$3000 (Collins, 2002)
Digital Logic	SM586 based on ZFx86 from ZF Micro Devices	133 MHz, 16 MB RAM MIN, USB, FDD, MSE, KBD, LPT, SP, CRT, +5, highly compact	5W power consumption, excessive features, complete system requires additional components	$400 (ea BB)

Arcom Control Systems	Pegasus	AMD Elan SC520 133 MHz, 16 MB RAM, 16 MB FLASH, EN, SP, LPT, MSE, KBD, FDD, HDD, RAM	No USB, larger than PC104, excessive power consumption, features	$400 (ea) Win CE Dev Kit ($1000)
Analog Devices	EZKIT	40 to 80 MHz DSP, ANSI C IDE, RAM, SP, ADC, DIO	No USB, limitations on system use on low-cost version	(free donation to department)
Other companies	Sensoray, ZFMicro Devices, Micronix (PC104 expansion cards A/D, power supply), Arbor, Advanced Digital Logic			(Not included because product profiles were very similar to those above)
Abbreviations	FDD = Floppy Disk Drive, HDD = Hard Disk Drive, SP = Serial Port, LPT = Parallel Port, DIO = Digital IO, AIO = Analog IO, KBD = keyboard port, MSE = mouse port, CRT = video monitor support, IDE = integrated development environment, MIN = minimum, +5 = single 5 VDC power supply, BB = bare board, no options or expanded memory, EN = Ethernet, LP = particularly low power consumption, RTC = real-time clock			

A pre-fabricated platform suffers drawbacks as well – products of the goals of mass production. A pre-fabricated design must maximize the needs it can meet for a wide variety of consumers and thus will usually include several unneeded components or lack specific needed components. For example, several PC104 boards include cost-inflating peripherals such as multiple serial and parallel ports and cathode ray tube (CRT) monitor ports, but lack universal serial bus (USB) support. The reason most likely lies in the design and optimization of a line of products that benefit from tested

and slightly dated PC technology brought to market to meets the needs of industrial design, an arena that strives to maximize the longevity of its initial capital investment, in addition to the fact that slower processors tend to consume less power and require less heat removal. In the case of platforms that retain many qualities of the PC, the time required to load an operating system delays the time at which the system is ready upon power on.

Thus it appears the challenge is, if possible, to select a pre-fabricated platform for embedded use that most closely meets all the needs of the project without incurring the cost of excessive peripherals. If all the needed functionality is not present on the platform, several supporting components will need to be fused with the platform, such as a USB controller module that will in itself require the acquisition of additional prototyping tools, negating the benefits of using a pre-fabricated platform. Such an investment might be better suited to future product application-specific streamlining. With these concerns in mind, several options present themselves.

Zworld offers a line of products that might be classified as simplified single-board computers especially suited to embedded products. These device make provisions for Ethernet or serial IO, ADC, display output and data processing. Most boards typically run at or around several tens of MHz. These products are simplified in that they do not include excessive peripheral features characteristic of PCs. They are programmable using a Zworld version of C similar to ANSI C, at a cost. There are no USB-equipped devices and the required data rates may require external interface chips to buffer the stored data. These devices appear to be better suited to lower rate sample gathering and control applications rather than signal processing.

Analog Devices has provided two EZKIT digital signal processing (DSP) evaluation platforms. These excel in their ability to process the APD-SA waveforms in as near to real time as possible, implementing hardware circular buffer pointers and a myriad of other boons for continuous signal processing. These kits are ANSI C programmable and retail for nearly $200 each. These kits are restricted in that they only access one-quarter of the platform's memory and lack USB or Ethernet capabilities. Thus, although inexpensive and powerful, early testing found the memory limitation to prevent adequate software development and debugging because the compiled debugging executable exceeded the memory limitation. Further, any IO routine would require software emulation or the addition of USB or similar IO controller peripherals, an additional cost. The cost of the full development platform rights, *i.e.* the rights to all the memory, is near $2000 dollars.

The implementation of PC104 platforms was examined. These platforms most often included extra peripherals but most closely approximated PCs, allowing for the use of the department's Visual Studio software in the APD-SA development. In most cases, these platforms lacked USB support but contained sufficient memory and storage and fast processors, relative to the requirements for the task, *e.g.* 486 and 586 Intel CPUs. These platforms would, in the less equipped and less expensive cases, require the addition of peripheral USB control. In general these platforms, even those more expensive with the proper USB or other IO capabilities suffered from excessive components for the APD-SA application and would be a source of wasted funds for each future APD-SA. Further, these platforms use typical PC operating systems and would necessitate start-up waits. Since they embody a compact PC platform, PC104

boards typically lack low-power consumption designs, some intended more for standard ATX power supply schemes, and other still consuming several Watts, in reduced-power consumption models.

Pre-fabricated platforms targeted for hand held devices offer certain benefits over industrial control oriented platforms, DSP development kits and PC104 boards: they are designed for minimal power consumption, more recently updated IO options including USB and Ethernet, small footprint and a minimum of extra peripheral components. These platforms most likely include more up-to-date IO options because they are aimed at the current mass entertainment consumption market. Since these platforms are smaller, they are tailored for smaller, faster operating system load times and standby modes while retaining the benefits associated with using pared-down versions of programming languages for which the department already owns licenses. Once installed and configured, WinCE 3.0 is instant-on with battery back-up. Most platforms arrive in a package loaded with an operating system that has already been configured with drivers for the platform's particular peripherals and a port for digital IO. Although hand held oriented platforms may still contain unneeded features and present some OS-load latency, the magnitude of these issues is smaller compared with other platforms.

The extended features of hand held oriented platforms can be implemented to minimize chip count and hence assembly time. One hand held oriented board, the CerfBoard, incorporates an Intel ARM CPU that integrates many functions into a single chip, minimizing board size, but retaining processing speed up to 233 MHz as of this writing. Since the prototype software was tested on a 233 MHz desktop PC, it would be

expected that few if any latency issues would arise when the software was transferred to a new platform. Since the platform runs at such a high speed, the need for creating any peripheral memory buffers is eliminated – these buffers can instead be implemented in software. Along with the elimination of such a memory buffer chip, the timing, address incrementing and memory control hardware is also eliminated. The platform includes USB and Ethernet eliminating the need for external component interface. Running WinCE or Embedded Linux, the software development tools will be free; Microsoft provides Embedded Visual Studio free. Linux operating systems are free except those offered as pre-compiled or ready-to-be-compiled packages at minimal cost ($50).

There are drawbacks to these platforms. Since the platforms are aimed at the portable entertainment or personal digital assistant market, manufacturers are more likely to discontinue support in efforts to stay competitive with cutting edge technology. Further, using an embedded operating system such as WinCE introduces complications in accessing low-level chip functions that could be used for APD-SA function.

APPENDIX I: PC HOSTED PROTOTYPE

To meet the objectives, the essential components and data flow for the APD were outlined and then reflected in preliminary software and hardware designs. The user interface was required to be as simple as possible, implementing as few control buttons as needed and making the device accessible to any user. Considerations also included the type of display and the types of interface connections for computer interface. Algorithms necessary for data analysis were evaluated in preparation for implementing a revision of the APD prototype on the hardware platform selected for the device. The MCU hardware selection process considered the memory required for processing the data, the required sampling rate, input and output data storage and communications protocol. PSpice student edition was used to a limited extent to simulate some basic electronics functions.

Once the hardware and software design was outlined, a PC was employed to test the firmware C code routines and ADC data collection electronics. Code was developed using MS Visual C++, an IDE that includes time profiling capabilities. Figure I.1 summarizes the prototype system and shows where the PC functions would be eventually replaced by an embedded MCU. The test code communicated with the data collection hardware through the computer parallel port. Since the development PC was a Windows 98 platform, the parallel port could be freely accessed as memory-mapped address. An adapter cable was used to transfer parallel port signals to a breadboard containing the ADC, motor control and 4-digit 7-segment LED display

circuitry. An adapter cable connected the breadboard to the APD100 body. Figure I.2 is a schematic of the prototype circuit. Figure I.3 diagrams the signal connections to the PC parallel port. A single push-button on the breadboard controlled all modes including calibration span, calibration zeroing, display of raw data and display of RR measurements.

The purpose of development on the PC was to substitute the PC CPU for an embedded processor in order to determine routine timing, RR measurement accuracy, ADC control procedures and APD-SA functionality. The prototype was first developed to drive the simplified 4-digit 7-segment LED display as a fast-response device. The fast-response device filled a buffer of RR measurements and at each new measurement updated the display with the weighted average of the most recently detected RR values. Functions were tuned to ensure that they could be executed within the sample interval. Data was logged to ensure that ADC values were sensible and that the RR calculation routines performed as required. Logged data and digital filter routines were analyzed in Matlab. Functions were tested and tuned to make common APD functions, such as RR measurement and rezeroing easily attained by short button pushes. The code was developed to allow calibration and diagnostic data reading upon longer button pushes. Figure I.4 documents the prototype operating modes. Figure I.5 documents the overall RR measurement code in the prototype. The code displayed text and data through the LED display digits.

The PC-driven prototype was then reviewed to determine the suitability of fast-response, the appearance of LED display digits, its use of push-button modes and its potential embedded MCU.

User Interface
Device Control
Mechanical
PC Host: Augmented Functions
(Plotting functions, graphic displays)
USB, Serial
8.8.8.8
Push-button
CPU
Final: Embedded WinCE/ Linux
Proto: Windows PC P233
SPI, 3-wire
Display Control IC
(MAX7221)
Tasks:
- Timing
- IC control
- Data processing
- IO management
SPI, 3-wire
Digital to Analog
PWM Signal
On/Off
PWM Control
(TL494)
Analog to Digital
(ADS7824/5)
Buffer
(MC33274)
Current Sense
Motor
Mouth
Pressure
Flow
Mouth
Pneumotach

Figure I.1. Prototype tested with a desktop computer performing the functions of the future embedded CPU components

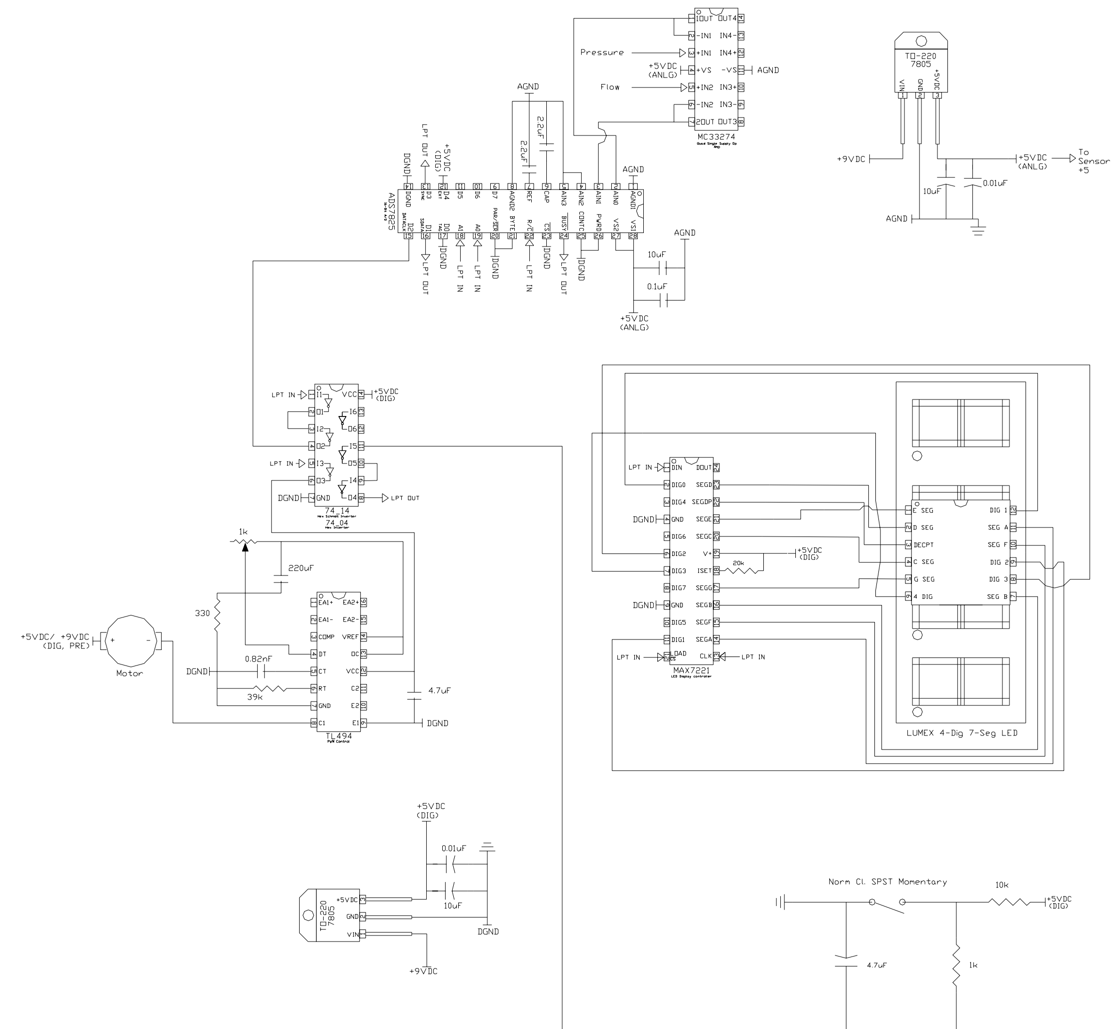


Figure I.2. APD-SA display, ADC and motor control circuit schematic in the PC-hosted prototype; this schematic incorporates CAD-drawn IC shapes that are scaled to actual sizes; electrical connections are labeled; any electrical connections with identical names are electrically connected even if they are not linked with a trace on the drawing

LPT Port (Computer) Function & Bit No. (Base 0)			LPT DB25 Pin No. (Base 1)	Breadboard Hole No. (Base 1)	Circuit Use
DATA:	D7:	Bit 7 (0x80)	9	17	
DATA:	D6:	Bit 6 (0x40)	8	16	
DATA:	D5:	Bit 5 (0x20)	7	15	ADS7824: CHADDR MSB A1 Pin 18
DATA:	D4:	Bit 4 (0x10)	6	14	ADS7824: CHADDR LSB A0 Pin 19
DATA:	D3:	Bit 3 (0x08)	5	13	
DATA:	D2:	Bit 2 (0x04)	4	12	MAX7221: DATAIN: DIN Pin 1
DATA:	D1:	Bit 1 (0x02)	3	11	MAX7221: CLOCK: CLK Pin 13
DATA:	D0:	Bit 0 (0x01)	2	10	MAX7221: LOAD(!CS): Pin 12
CONTROL:	!STROBE:	Bit 0 (0x01)	1	9	
CONTROL:	!SEL_INP:	Bit 3 (0x08)	17	8	ADS7824: DATACLK: Pin 15**
CONTROL:	INIT:	Bit 2 (0x04)	16	7	ADS7824: R/!C: Pin 22
STATUS:	!ERROR:	Bit 3 (0x08)	15	6	
CONTROL:	!AUTOFEED:	Bit 1 (0x02)	14	5	TL494: PWM Motor Enable Pin 12*
STATUS:	SELECT:	Bit 4 (0x10)	13	4	BUTTON !Pressed***
STATUS:	PE:	Bit 5 (0x20)	12	3	ADS7824: SDATA: Pin 16
STATUS:	BUSY:	Bit 7 (0x80)	11	2	ADS7824: !BUSY: Pin 24
STATUS:	!ACK:	Bit 6 (0x40)	10	1	ADS7824: SYNC: Pin 13

Notes: Hardware bit inversion in LPT port convention requires the following bit XOR masks for software output:
DATA: none
CONTROL: ^0x0b
STATUS: ^0x80
*Single 74HCT14 buffered inversion **Double 74HCT14 buffered inversion ***Debounced, double 74HCT14 buffered inversion

Figure I.3. Parallel port to breadboard connections for IC control and data acquisition in the APD-SA PC hosted prototype

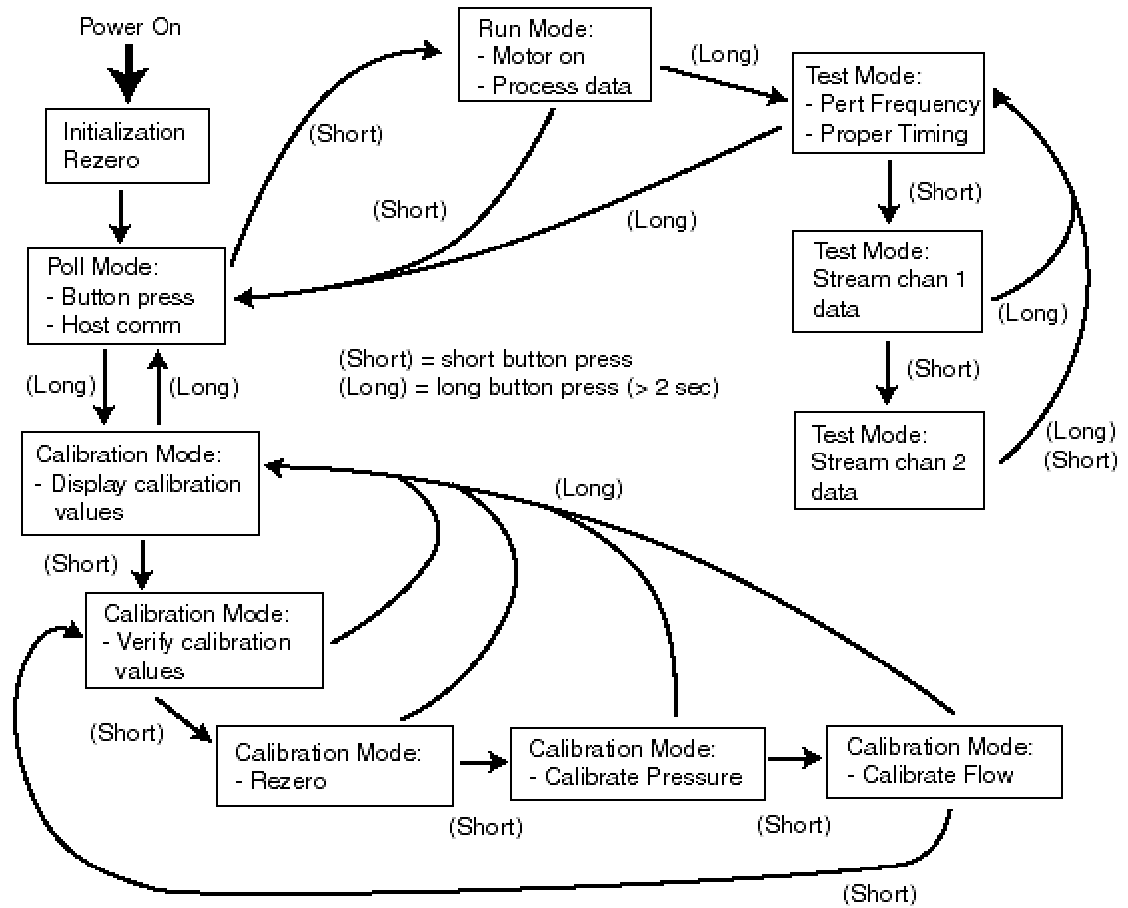


Figure I.4. APD-SA PC-hosted prototype mode flow chart indicating how short and long button presses change mode

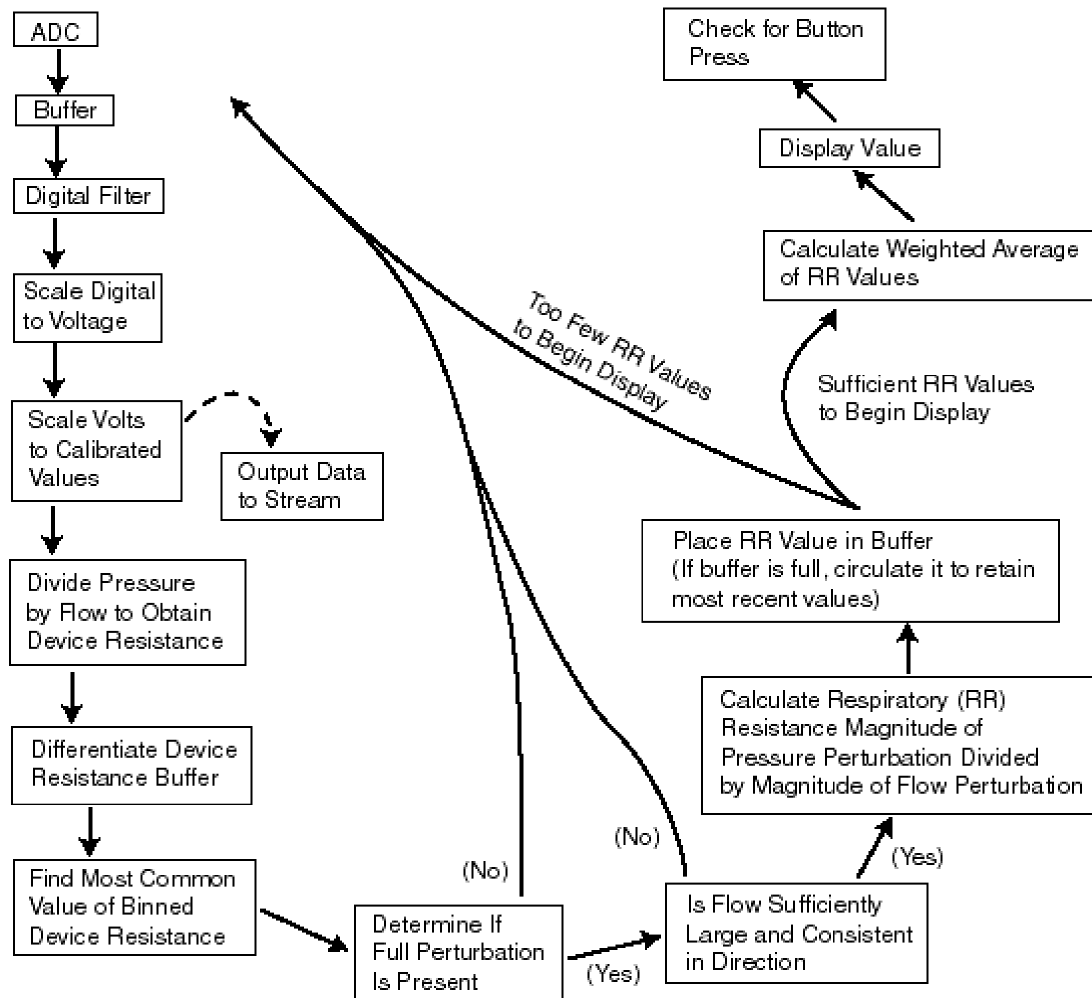


Figure I.5. Software flow chart for APD-SA PC hosted prototype indicating processes that occur with each clock tick

REFERENCES

Akay, M. and J. A. Daubenspeck. 2000. Respiratory related evoked responses to graduated pressure pulses using wavelet transform methods. *Annals of Biomedical Engineering.* 28:1126-1135.

American Lung Association. 2001. Trends in Asthma Morbidity and Mortality.

American Thoracic Society. 1998. Dyspnea: Official statement adopted by ATA board of directors. *American Journal of Respiratory Critical Care and Medicine*. 159:321-340.

American Thoracic Society. 1999. Pulmonary rehabilitation. *American Journal of Respiratory Critical Care and Medicine*. 159:1666-1682.

Axelson, J. 2001. USB Complete. Madison, Wisconsin: Lakeview Research.

Axelson, J. 1998. Serial Port Complete. Madison, Wisconsin: Lakeview Research.

Axelson, J. 1997. Parallel Port Complete. Madison, Wisconsin: Lakeview Research.

Barnas, G. M., N. C. Heglund, D. Yager, K. Yoshino, S. H. Loring and J. Mead. 1989. Impedance of the chest wall during sustained respiratory muscle contraction. *Journal of Applied Physiology.* 28:365-372.

Barnas, G. M., K. Yoshino, J. Fredberg, Y. Kikuchi, S. H. Loring, and J. Mead. 1990. Total and local impedance of the chest wall up to 10 Hz. *Journal of Applied Physiology.* 68:1409-1414.

Bates, J. H., P. Baconnier and J. Milic-Emili. 1988. A theoretical analysis of interrupter technique for measuring respiratory mechanics. *Journal of Applied Physiology.* 64(5): 2204-2214.

Bouhuys A. and B. Jonson. 1967. Alveolar pressure, airflow rate, and lung inflation in man. *Journal of Applied Physiology.* 22:1086-1100.

Burns, C. B., W. R. Taylor and R. H. Ingram, Jr. 1985. Effects of deep inhalation in asthma: relative airway and parenchymal hysteresis. *Journal of Applied Physiology*. 59:1590–1596.

Carrieri-Kohlman, V., J. M. Gormley, M. K. Douglas, S. M. Paul and M. S. Stulbarg. 1996. Exercise training decreases dyspnea and the distress and anxiety associated with it: monitoring alone may be as effective as coaching. *Chest.* 110:1526 -1535.

Celli, B. R. 1995. Pulmonary rehabilitation in patients with COPD. *American Journal of Respiratory Critical Care and Medicine*. 152(3):861-864.

Cosio, M. and A. Guerassimov. 1999. Chronic obstructive pulmonary disease: inflammation of small airways and lung parenchyma. *American Journal of Respiratory Critical Care and Medicine*. 160:S21-S25.

Crapo, R.O. and R.L. Jensen. 2001. Test report: QRS Diagnostic, LLC. Sensaire. Salt Lake City, Utah: Intermountain Health Care: LDS Hospital.

Dirksen, A., N. H. Holstein-Rathlou, F. Madsen, L. T. Skovgaard, C. S. Ulrik, T. Heckscher and A. Kok-Jensen. 1998. Long-range correlations of serial FEV1 measurements in emphysematous patients and normal subjects. *Journal of Applied Physiology*. 85(1):259-265.

Ducharme, F. M., G. M. Davis and G. R. Ducharme. 1998. Pediatric reference values for respiratory resistance measured by forced oscillation. *Chest*. 113: 1322-1328.

Ducharme, F. M. and G. M. Davis. 1997. Measurement of respiratory resistance in emergency departments: feasibility in young children. *Chest.* 111:1519-1525.

Eid, N., B. Yandell, L. Howell, M. Eddy and S. Sheikh. 2000. Can peak expiratory flow rate predict airflow obstruction in children with asthma? *Pediatrics*. 105(2):354-358.

Fleming, H. E., F. F. Little, D. Schnurr, P. C. Avila, H. Wong, J. Liu, S. Yagi and H. A. Boushey. 1999. Rhinovirus-16 colds in healthy and in asthmatic subjects. *American Journal of Respiratory Critical Care and Medicine*. 160(1):100-108.

Frank, N. R., J. Mead and B. G. Ferris. 1957. The mechanical behavior of the lungs in healthy elderly persons. *Journal of Clinical Investigations.* 36:1680–1687.

Gern, J. E., W. Calhoun, C. Swenson, G. Shen and W. W. Busse. 1997. Rhinovirus infection preferentially increases lower airway responsiveness in allergic subjects. *American Journal of Respiratory Critical Care and Medicine*. 155(6):1872-1876.

Goldsmith, T. 2002. National Institute for Occupational Safety and Health, Health Effects Laboratory Division, Engineering and Control Technology Branch, personal communication, September 29, 21st Southern Biomedical Engineering Conference.

Higgins, M. 1993. Epidemiology of obstructive pulmonary disease. In *Principles and Practice of Pulmonary Rehabilitation,* ed. R. Cassaburi and T. L. Petty. Philadelphia: W. B. Saunders.

Hernandez, M. T., T. M. Rubio, F. O. Ruiz, H. S. Riera, R. S. Gil and J. C. Gómez. 2000. Results of a home-based training program for patients with COPD. *Chest*. 118:106-114.

Herxheimer, H. 1975. *A guide to bronchial asthma*. New York, NY: Academic Press.

Jacob, S. V., L. C. Lands, A. L. Coates, G. M. Davis, C. F. MacNeish, L. Hornby, S. P. Riley and E. W. Outerbridge. 1997. Exercise ability in survivors of severe bronchopulmonary dyplasia. *American Journal of Respiratory Critical Care and Medicine*. 155(6):1925-1929.

Janson-Bjerklie, S. and S. Shnell. 1988. Effect of peak flow information on patterns of self-care in adult asthma. *Heart and Lung*. 17(5):543-549.

Johnson, A. T., C. S. Lin and J. N. Hochheimer. 1984a. Airflow perturbation device for measuring airways resistance of humans and animals. *IEEE Transactions of Biomedical Engineering*. 9:622-626.

Johnson, A. T., J. N. Hochheimer and J. Windle. 1984b. Airflow perturbation device for respirator research and medical screening. *Journal of the ISRP*. 2:338-346.

Jones, K. P. and M. A. Mullee. 1990. Measuring peak expiratory flow in general practice: comparison of mini Wright peak flow meter and turbine spirometer. *British Medical Journal*. 300(6740):1629-1631.

Kessler, V., G. Mols, H. Bernhard, C. Haberthur and J. Guttmann. 1999. Interrupter airway and tissue resistance: errors caused by valve properties and respiratory system compliance. *Journal of Applied Physiology*. 87(4):1546–1554.

Khan, J. H., S. Magnetti, E. Davis and J. Zhang. 2000. Late outcome of open heart surgery in patients 70 years or older. *Annals of Thoracic Surgery*. 69:165-170.

Klaustermeyer, W. B., M. Kurohara and G. A. Guerra. 1990. Predictive Value of Monitoring Expiratory Peak Flow Rates in Hospitalized Adult Asthma Patients. *Annals of Allergy*. 64(3):281-284.

Krell, W. S., K. P. Agrawal and R. E. Hyatt. 1984. Quiet breathing vs. panting methods for determination of specific airway conductance. *Journal of Applied Physiology.* 57(6): 1917-1922.

Lausted, C. G. 1997. Development of the Airflow Perturbation Device. M.S. Thesis. Biological Resources Engineering Dept., University of Maryland at College Park.

Lausted, C. G. and A. J. Johnson. 1998. Respiratory resistance measured by an airflow perturbation device. *Physiological Measurements*. 20(1999):21-35.

Lehtola, P. J. 1986. Improving the Airflow Perturbation Device. M.S. Thesis. Agricultural Engineering Dept., University of Maryland, College Park.

Leidy, N. K., K. S. Chan and C. Coughlin. 1998. Is the asthma quality of life questionnaire a useful measure for low-income asthmatics? *American Journal of Respiratory Critical Care and Medicine*. 158:1082-1090.

Lemes, L. N. A. and P. L. Melo. 2003. Forced oscillation technique in the sleep apnoea/ hypopnoea syndrome: identification of respiratory events and nasal continuous positive airways pressure titration. *Physiological Measurement.* 24(2003):11-25.

Malmberg, L.P., S. Mieskonen, A. Pelkonen, A. Kari, A. R. A. Sovijarvi and M. Turpeinen. 2000. Lung function measured by the oscillometric method in prematurely born children with chronic lung disease. *European Respiratory Journal.* 16:598-603.

Martinez, F. J., M. M. de Oca, R. I. Whyte, J. Stetz, S. E. Gay and B. R. Celli. 1997. Lung-volume reduction improves dyspnea, dynamic hyperinflaction, and respiratory muscle function. *American Journal of Respiratory Critical Care and Medicine*. 155(6): 1984-1990.

Martinez, T. Y. 2000. Evaluation of the Short-Form 36-Item questionnaire to measure health-related quality of life in patients with idiopathic pulmonary fibrosis. *Chest*. 117:1627-1632.

Mazumder, D. N. G., R. Haque, N. Ghosh, B. K. De, A. Santra, D. Chakraborti and A. H. Smith. 2000. Arsenic in drinking water and the prevalence of respiratory effects in West Bengal, India. *International Journal of Epidemiology*. 29(6):1047-1052.

Mead, J. and J. L. Whittenberger. 1953. Properties of the human lungs measured during spontaneous respiration. *Journal of Applied Physiology*. 5:779-796.

Melo, P. L. and L. N. A. Lemes. 2002. Instrumentation for the analysis of respiratory system disorders during sleep: design and application. *Review of Scientific Instruments*. 73(11):3926-3932.

Miller, M. R., O. F. Pedersen and T. Sigsaard. 1997. Spirometry with a Fliesch pneumotachograph: upstream heat exchanger replaces heating requirement. *Journal of Applied Physiology*. 82(4):1053-1057.

Mitzner, W. and R. H. Brown. 2000. Potential mechanism of hypperresponsive airways. *American Journal of Respiratory Critical Care and Medicine*. 161(5):1619-1623.

National Heart, Lung and Blood Institute. 1997. Data Fact Sheet on Asthma Statistics. Bethesda, MD: National Institute of Health. Publication 55-798.

National Heart, Lung and Blood Institute. 2002. Morbidity and Mortality: 2002 Chart Book on Cardiovascular, Lung, and Blood Diseases. Bethesda, MD: National Institute of Health.

Navajas, D. and R. Farre. 2001. Forced oscillation assessment of respiratory mechanics in ventilated patients. *Critical Care.* 5:3-9.

Nicholson, K. G., J. Kent and D. C. Ireland. 1993. Respiratory viruses and exacerbations of asthma in adults. *British Medical Journal*. 307(6910):982-986.

Obase, Y., T. Shimoda, K. Mitsuta, H. Matsuse and S. Kohno. 2000. Two patients with occupational asthma who returned to work with dust respirators. *Occupational Environmental Medicine*. 57:62-64.

Olton, D. S. and A. R. Noonberg. 1980. *Biofeedback: Clinical Applications in Behavioral Medicine*. Englewood Cliffs, New Jersey: Prentice-Hall.

Oud, M., E. H. Dooijes and J. S. van der Zee. 2000. Asthmatic airways obstruction assessment based on detailed analysis of respiratory sound spectra. *IEEE Transactions on Biomedical Engineering*. 47(11):1450-1455.

Quanjer, P. H, M. D. Lebowitz, I. Gregg, M. R. Miller and O. F. Pedersen. 1997. Peak expiratory flow: conclusions and recommendations of a Working Party of the European Respiratory Society. *European Respiration Journal*. 24:2s-8s.

Rakes, G. P., E. Arruda, J. M. Ingram, G. E. Hoover, J. C. Zambrano, F. G. Hayden, T. A. E. Platts-Mills and P. W. Heymann. 1999. Rhinovirus and respiratory syncytial virus in wheezing children requiring emergency care. *American Journal of Respiratory Critical Care and Medicine*. 159(3):785-790.

Ramirez, J., I. Leon and LM Rivera. 1986. Episodic laryngeal dyskinesia. clinical and psychiatric characterization. *Chest*. 90:716-721.

Silverman, N. S., A. T. Johnson, W. S. Scott, F. Koh. 2002. Exercise-induced respiratory resistance changes measured by airflow perturbation device. *Applied Ergonomics.* In press.

Snow, M. G. 1997. Airway resistance measurements. In: *Respiratory Clinics of North America,* eds. A. Adams and C. McArthur. Philadelphia: W. B. Saunders.

Snow, M. G. 2000. Airways resistance measurements in the evaluation of obstructive lung disease. *American Association for Respiratory Care Times.* 24(1):48-51.

Snow, M. G., B. Anderson, K. Kandal and R. J. Fallat. 1995. Airways resistance and lung volume are valuable adjuncts to spirometry for assessing reversible airways obstruction. *Respiratory Care.* 40(11):1179-1181.

Snowden, C. P., T. Hughes, J. Rose and D. R. Roberts. 2000. Pulmonary edema in patients after liver transplantation. *Liver Transplantation*. 6(4):466-470.

Stahlman, J. E. and L. M. Salmun. 1999. New developments in the home monitoring of asthma. *The Internet Journal of Asthma, Allergy and Immunology.* 1(1).

Stein, M., G. Tanabe, V. Rege and M. Khan. 1966. Evaluation of spirometric methods to assess abnormalities in airways resistance. *American Review of Respiratory Disease.* 93(2):257-263.

Trigg, C. J., K. G. Nicholson, J. H. Wang, D. C. Ireland, S. Jordan, J. M. Duddle, S. Hamilton and R. J. Davies. 1996. Bronchial inflammation and the common cold: a comparison of atopic and non-atopic individuals. *Clinical and Experimental Allergy*. 26 (6):665-676.

United States Centers for Disease Control. 1998. Forecasted State-Specific estimates of Self-Reported Asthma Prevalence. *Morbidity and Mortality*. 47:1022-1025.

Verbeken, E. K., M. Cauberghs and K. P. Van De Woestijne. 1996. Membranous bronchioles and connective tissue network of normal and emphysematous lungs. *Journal of Applied Physiology*. 81(6):2468-2480.

Vollmer, W. M., L. E. Markson, E. O'Connor, E. A. Frazier, M. Berger and A. S. Buist. 2002. Association of asthma control with health care utilization and quality of life. *American Journal of Respiratory and Critical Care Medicine.* 165(2):195-199.

Widdicombe, J. G. 1989. Nervous receptors in the tracheobronchial tree: airway smooth muscle reflexes. In *Airway smooth muscle in health and disease*, ed. R. F. Coburn. 35-53. New York, NY: Plenum.

Wijkstra, P. J., T. W. van der Mark, J. Kraan, R. van Altena, G. H. Koeter and D. S. Postma. 1996. Long-term effects of home rehabilitation on physical performance in chronic obstructive pulmonary disease. *American Journal of Respiratory Critical Care and Medicine*. 153(4):1234-1241.

Wojnarowski, C., I. Eichler, C. Gartner, M. Gotz, S. Renner, D. Y. Koller and T. Frischer. 1997. Sensitization to Apsergillus fumigatus and lung function in children

with cystic fibrosis. *American Journal of Respiratory Critical Care and Medicine*. 155 (6):1902-1907.

Wong, C. M., T. H. Lam, J. Peters, A. J. Hedley, S. G. Ong, A. Y. Tam, J. Liu and D. J. Spiegelhalter. 1998. Comparison between two districts of the effects of an air pollution intervention on bronchial responsiveness in primary school children in Hong Kong. *Journal of Epidemiology and Community Health.* 52:571-578.

Wright, B. and C. McKerrow. 1959. Maximum forced expiratory flow rate as a measure of ventilatory capacity with a description of a new portable instrument for measuring it. *British Medical Journal.* 2:1041-1047.

Wu, M. T., J. M. Chang, A. A. Chiang, J. Y. Lu, H. K. Hsu, W. H. Hsu and C. F. Yang. 1994. Use of quantitative CT to predict postoperative lung function in patients with lung cancer. *Radiology*. 191:257-262.

Yeh, M. P., R. M. Garner, T. D. Adams and F. G. Yanowitz. 1982. Computerized determination of pneumotachometer characteristics using a calibrated syringe. *Journal of Applied Physiology*. 53(7):280-285.