LA-UR-14-27446
Approved for public release; distribution is unlimited.

Title: Final Report: Part 1. In-Place Filter Testing Instrument for Nuclear Material Containers. Part 2. Canister Filter Test Standards for Aerosol Capture Rates.

Author(s): Brown, Austin Douglas
Runnels, Joel T.
Moore, Murray E.
Reeves, Kirk Patrick

Intended for: Report

Issued: 2014-11-02 (rev.1)

LA-UR-14-27446
Approved for public release; distribution is unlimited.

Title: Final Report: Part 1. In-Place Filter Testing Instrument for Nuclear Material Containers. Part 2. Canister Filter Test Standards for Aerosol Capture Rates.

Author(s): Brown, Austin Douglas
Runnels, Joel T.
Moore, Murray E.
Reeves, Kirk Patrick

Intended for: Report

Issued: 2014-09-23

LA-UR-14-

Approved for public release; distribution is unlimited.

Title: Final Report: Part 1. In-Place Filter Testing Instrument for Nuclear Material Containers. Part 2. Canister Filter Test Standards for Aerosol Capture Rates.

Author(s): A.D. Brown, J.T. Runnels, M.E. Moore, K. Reeves

Intended for: Chris Chaves
Project Manager
Nuclear Safety R&D Program
Office of Nuclear Safety (AU-30)

LAUR-14-27446 Final Report: WAS Project No.: 2013-HS-2013008:

I. In-Place Filter Testing Instrument for Nuclear Material Containers

II. Canister Filter Test Standards for Aerosol Leak Rates

A.D. Brown, J.T. Runnels, M.E. Moore, K. Reeves

Los Alamos National Laboratory, Los Alamos, NM 87544, USA

This project was funded by the US Department of Energy Health, Safety and Security Program, WAS Project No.: 2013-HS-2013008, B&R No. HU1004500.

Abstract

A portable instrument has been developed to assess the functionality of filters and o-rings on nuclear material storage canisters, without requiring removal of the canister lid. Additionally, a set of fifteen filter standards were procured for verifying aerosol leakage and pressure drop measurements in the Los Alamos Filter Test System.

The US Department of Energy uses several thousand canisters for storing nuclear material in different chemical and physical forms. Specialized filters are installed into canister lids to allow gases to escape, and to maintain an internal ambient pressure while containing radioactive contaminants. Diagnosing the condition of container filters and canister integrity is important to ensure worker and public safety and for determining the handling requirements of legacy apparatus. This report describes the In-Place-Filter-Tester, the Instrument Development Plan and the Instrument Operating Method that were developed at the Los Alamos National Laboratory to determine the "as found" condition of unopened storage canisters. The Instrument Operating Method provides instructions for future evaluations of as-found canisters packaged with nuclear material. Customized stainless steel canister interfaces were developed for pressure-port access and to apply a suction clamping force for the interface. These are compatible with selected Hagan-style and SAVY-4000 storage canisters that were purchased from NFT (Nuclear Filter Technology, Golden, CO).

Two instruments were developed for this effort: an initial Los Alamos POC (Proof-of-Concept) unit and the final Los Alamos IPFT system. The Los Alamos POC was used to create the Instrument Development Plan: (1) to determine the air flow and pressure characteristics associated with canister filter clogging, and (2) to test simulated configurations that mimicked canister leakage paths. The canister leakage scenarios included quantifying: (A) air leakage due to foreign material (i.e. dust and hair) fouling of o-rings, (B) leakage through simulated cracks in o-rings, and (C) air leakage due to inadequately tightened canister lids. The Los Alamos POC instrument determined pertinent air flow and pressure quantities, and this knowledge was used to specify a customized Isaac® (Zaxis, Salt Lake City, UT) leak test module. The final Los Alamos IPFT (incorporating the Isaac® leak test module) was used to repeat the tests in the Instrument Development Plan (with simulated filter clogging tests and canister leak pathway tests).

The Los Alamos IPFT instrument is capable of determining filter clogging and leak rate conditions, without requiring removal of the container lid. The IPFT measures pressure decay rate from 1.7E-03 inWC/sec to 1.7E-01 inWC/sec. On the same unit scale, helium leak testing of canisters has a range from 5.7E-07 inWC/sec to 1.9E-03 inWC/sec. For a 5-quart storage canister, the IPFT measures equivalent leak flow rates from 0.03 to 3.0 cc/sec. The IPFT does not provide the same sensitivity as helium leak testing, but is able to gauge the assembled condition of as-found and in-situ canisters.

LIST OF DEFINITIONS

Canister (Container). A stainless steel vessel designed for the long term storage of nuclear materials in various physical and chemical forms. The Hagan and SAVY canisters (NFT Inc.) have a filter integrated into the canister lid.

Canister Interface. A stainless steel fitting designed by Los Alamos to fit over the Hagan or SAVY canister filter. The canister interface has two concentric chambers, one measure the air pressure in the canister, the other chamber uses a suction force from a pump to clamp the canister interface onto the canister lid. (The Hagan interface is different from the SAVY interface.)

Hagan canister. A type of nuclear material storage canister, introduced in the 1980s. This canister includes a carbon fiber filter, and the lid requires about seven full turns for complete sealing, and the canister has been shown to leak after drop testing. The Hagan o-ring is mounted on the top lip edge of the cylindrical canister body, and easily falls out of its groove.

IDP (Instrument Development Plan). A variety of laboratory tests that mimic filter clogging and o-ring failure mechanisms. These were used to determine operating conditions for the initial Proof-of-Concept test device. They are not intended for field testing of canisters that contain nuclear materials.

IOM (Instrument Operating Method). A series of instructions for an IPFT (In-Place-Filter-Tester) device to field-test the integrity of storage canisters that are packaged with nuclear materials.

IPFT (In-Place-Filter-Tester). This is the final prototype device designed by the Los Alamos team, using quantitative input from the IDP (Instrument Development Plan). It used the same Canister Interfaces that were developed with the first POC device.

Isaac™ leak tester. A leak tester module manufactured by the Zaxis company, Salt Lake City, UT.

LANL. Los Alamos National Laboratory.

NCR. Non Compliance Report. In this report, this refers to a set of Hagan canister lids that are utilized by NFT Inc for standards testing.

NFT Inc. Nuclear Filter Technology Inc. (Golden CO) Manufacturer of Hagan and SAVY-4000 nuclear material storage canisters.

POC (Proof-of-Concept). In this report, it is the first in-place-filter-tester built by the Los Alamos design team. It is an automated device that determines if a canister filter is clogged, or if the o-ring leaks. It uses an Arduino™ microprocessor with LANL software to control air flows and to measure pressure quantities.

Pressure (air or gas). In this report, air (or gas) pressure is typically expressed in terms of the gas gauge pressure (e.g. inches of water column, inWC). A standard atmosphere is defined as 405 inches of water column (in WC), or as 101.1 kPa. The ambient air pressure at the Los Alamos test facility is 308 inWC.

SAVY canister (aka the "SAVY-4000"). A nuclear material storage canister introduced by NFT in 2011. It contains a non-fouling ceramic fiber filter, the lid is sealed with a bayonet mount (partial turn), and was shown to be leak-tight after drop tests. The SAVY o-ring is mounted in the canister lid, and does not readily fall out of its groove.

PART I.

IN-PLACE-FILTER-TESTER FOR NUCLEAR MATERIAL CONTAINERS

(1) INTRODUCTION

Nuclear Filter Technology (NFT Inc., Golden, CO) manufactures nuclear material storage canisters with high performance filters integrated into the lids of the canisters. In addition, NFT also manufactures filters for use with waste drums, and bag-out bags. Los Alamos National Laboratory uses several thousand containers to store nuclear material in various chemical formulations. The containers have an air filter (see Figure 1) mounted in the canister lid, and they employ an o-ring seal to maintain a leak-tight boundary. The goal of this project was to develop a portable In-Place-Filter-Tester instrument that will assess filter and o-ring functionality of a container, without requiring removal of the container lid. The instrument does not measure aerosol collection efficiency or capture, but can gauge filter performance and leak rate conditions from pressure drop measurements.

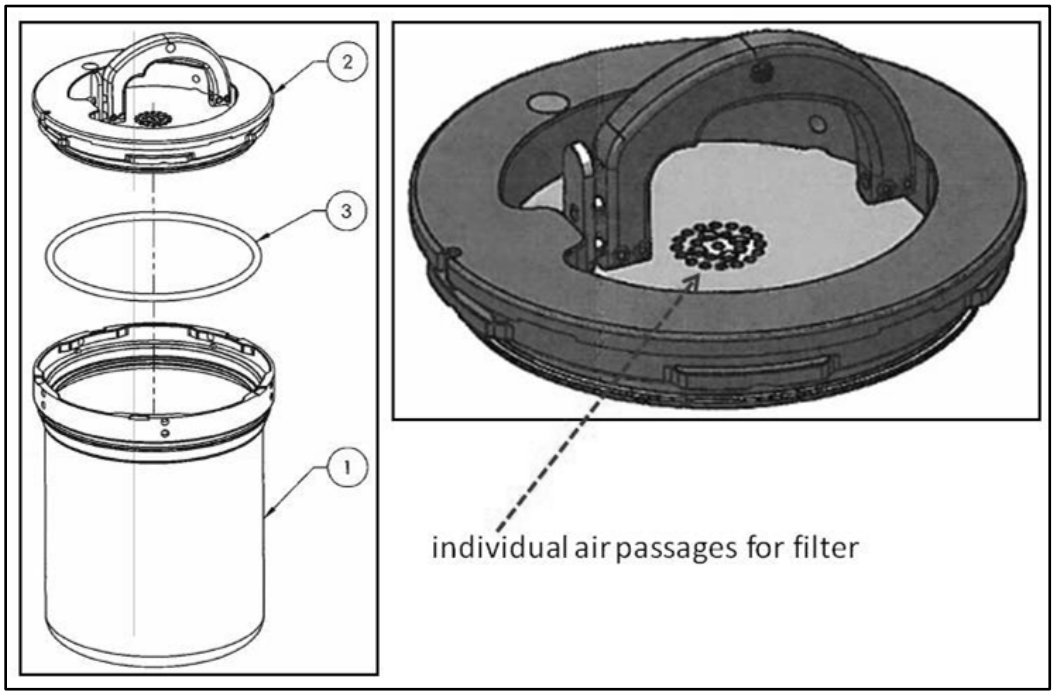


Figure 1. A SAVY storage canister from NFT Inc. (1) Canister (2) Lid. (3) O-ring. The lid close-up shows twenty five holes, 3 mm diameter, for filter air flow. The filter material is a 25 mm diameter disk, pressed between two plates which form the entire lid.

(2) BACKGROUND

The canister filters allow air (and hydrogen gas generated from radiolysis) to pass through a matrix of carbon or ceramic fibers, but the filters are designed to trap radioactive aerosol particles when air passes from the container interior through the filter (Moore ME et al 2011). The gas flow is driven by the

pressure force of hydrogen generated inside the containers and by pressure gradients due to atmospheric air pressure fluctuations.

In this report, filter performance will be described in terms of the "leak percent" (Table 1), that is, the amount of aerosol that passes through a filter (Moore ME et al 2011). The leak percent is similar to "aerosol penetration", but it is defined as a value measured over a range of particle sizes, whereas aerosol penetration is defined at discrete particle size diameters.

Table 1. Nomenclature to describe filter performance.

	Filter leak testing	**HEPA filters defined by ASME AG-1**
Aerosol Measurement Instrument	Single Channel (Photometer)	Multi-Channel Spectrometer
Test Aerosol	The aerosol concentration of a polydisperse distribution of sizes is measured by a single channel photometer (e.g. mean size of DOP oil at 0.45 μm).	Aerosol concentrations are measured at each individual particle size, i.e. "essentially monodispersed 0.3 μm" as mentioned in ASME AG-1.
	Leak of Aerosol = C_D/C_U	P = Penetration = C_D/C_U
	Capture of Aerosol = 1 - C_D/C_U	E = Efficiency = 1 - C_D/C_U

Canisters at Los Alamos (Figure 2) are designed to have a maximum normal operating pressure range of 0.25 to 0.50 inWC (1 to 2 kPa). In this report, air (or gas) pressure is typically expressed in terms of the gas gauge pressure (e.g. inches of water column, inWC). A standard atmosphere is defined as 405 inches of water column (in WC), or as 101.1 kPa. The ambient air pressure at the Los Alamos test facility is 308 inWC.

Sufficiently thin walls are implemented for content inspection, thus overpressure can result in visible deformation. Current designs have a "no-maintenance" life of five years, however, the desired life extension to a service life of 40 years or greater would substantially reduce laboratory resources. Studies of o-ring degradation indicate that the expected service life far exceeds five years, but service life limitations for both the o-ring and filter are unknown at this time. Surveillance of in-service canisters is necessary for this life-extension goal to ensure the ability to meet design criteria, and to detect potential problems that might be introduced during storage. Canister filters must deliver a minimum of 200 ml/min of air at no more than 1.0 inch water column (in WC) pressure differential, while capturing greater than 99.97% of 0.45 micron mean diameter aerosol. Current evaluation includes visual deformation inspection for canister overpressure caused by a clogged filter, and contamination surveys indicating possible o-ring seal failures (Anderson LL et al 2013).

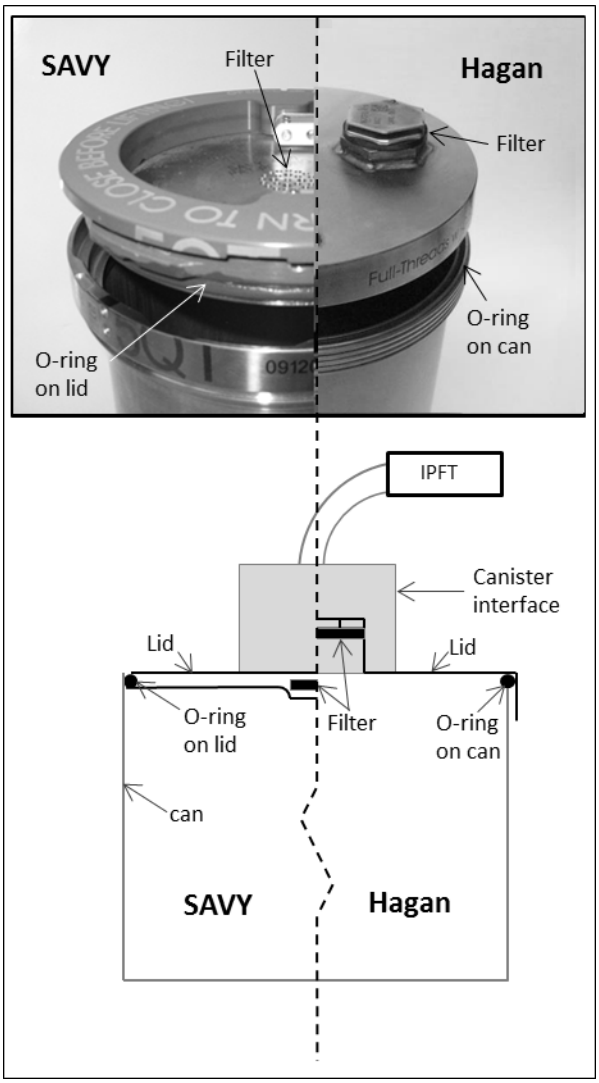


Figure 2. A side-by-side graphical comparison of the SAVY and Hagan containers.

The Hagan canister (Brassell GW and RP Brugger, 1985) was developed by NFT in the 1980s, and the new generation SAVY canister (Stone TA, Smith PH 2011) was recently introduced (Anderson LL et al 2013). Both the Hagan and SAVY canisters (Figure 2) are manufactured from stainless steel, but the new generation SAVY container has several improved safety and operational features (see Table 2 and Figure 2). The SAVY canister has a ceramic fiber filter that resists the chemical fouling issues that have been noticed (Havrilla GJ 2000, and Havrilla GJ and S Bowen 2002). Filter clogging of the older Hagan style canisters was investigated in these studies (Havrilla GJ 2000, and Havrilla GJ and S Bowen 2002) where they concluded that the clogging was not due to the test aerosol that was used to measure the filter penetration.

Table 2. Comparison of some features of the Hagan and SAVY canisters.

Feature	**Hagan canister (1980s design)**	**SAVY canister (2011 and beyond)**
Stainless steel construction	304	316
Filter fiber composition	Carbon fiber filter	Ceramic fibers (resists chemical fouling)
Number of turns to seal the lid	About seven turns of the lid	One-quarter of a turn (bayonet)
Integral lockpin	No	Yes
O-ring location	On canister	On lid
Filter location	Protruding above lid	Recessed and integrated into lid

Mechanical sealing of the Hagan canister depends upon the twist-tightening the lid onto the canister body, but the SAVY canister seal is maintained by the press-fit seal of the o-ring when the lid is pressed into place in the top of the canister. The SAVY container lid has an o-ring which slides into the container and does not rotate while the lid is positioned into its locked-tight location. The Hagan o-ring lies on the top of the can, and it may slide across the inner face of the lid while it is threaded multiple revolutions into a locked-tight location.

A list of additional differences between the Hagan and the SAVY canister (Anderson LL et al 2013) is available for reference.

Technical standards (ASTM 2013) and procedures (Moore ME 2013a) for leak testing have been developed and implemented where a vessel or a container is “charged” or “filled” with a positive pressure (or with a vacuum). When a predetermined pressure or vacuum is established (Figure 3), and after a defined time to allow the canister pressure to “settle”, the test system monitors the rate of change of the internal pressure of the canister.

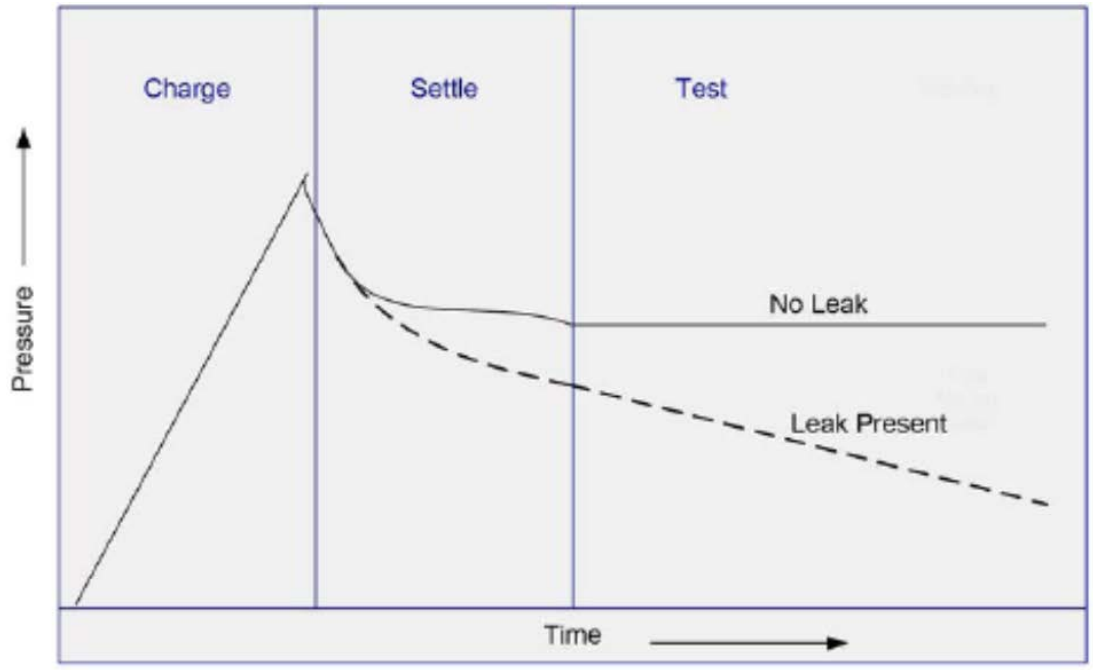


Figure 3. ASTM E2930 "Illustration of pressure decay test data."

(3) IPFT SYSTEM OPERATION SUMMARY

The Los Alamos IPFT (In-Place-Filter-Tester) is designed to assess the filter and o-ring functionality of nuclear material storage canisters, without the need to remove the container lid (Figure 4). To evaluate a canister, a customized canister interface (Figures 5 and 6) is placed on the top of the canister lid, above the canister filter.

Due to the different filter housings associated with the two lids, multiple canister interfaces were fabricated. They are both, however, geometrically equivalent for all testing purposes. The outer annulus is designed to provide a clamping force between the inerface, and the lid of the container, while the inner most volume applies a test vacuum directly to the filter being tested. These two testing ports are contained between 70 durometer o-rings which prevent cross flow. The recessed design of SAVY filters requires alignment pins to assure proper placement of the test fixture. The Hagan filter housing on the other hand, protrudes inside the measurement port and self-aligns its test fixture. The described interface features are portrayed below (Figure 5).

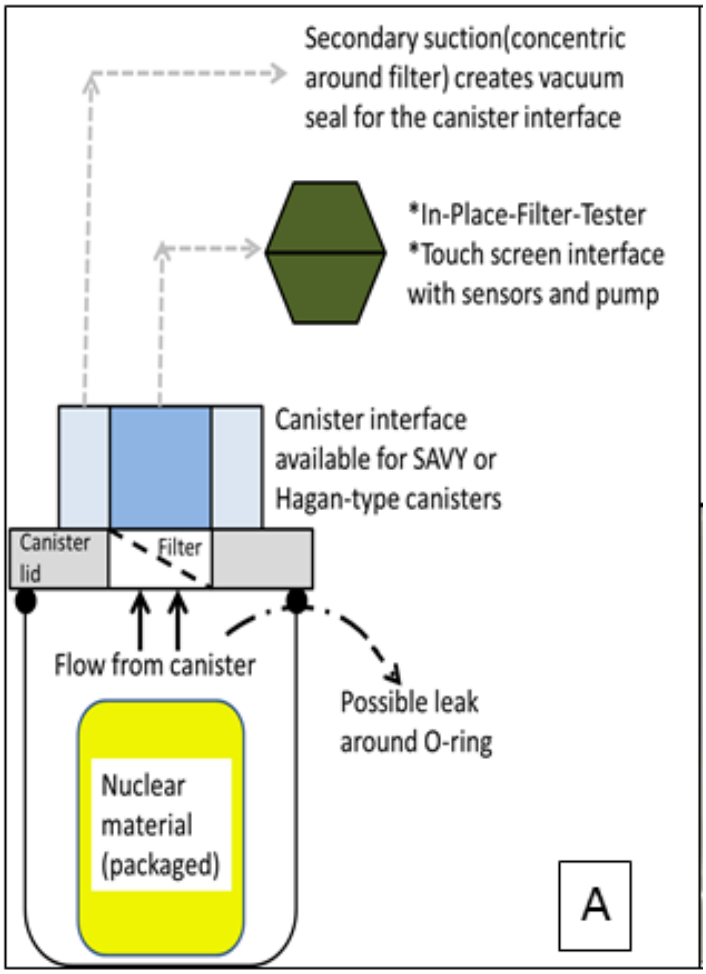


Figure 4. The IPFT (In-Place-Filter-Tester) assesses filter clogging or canister leakage

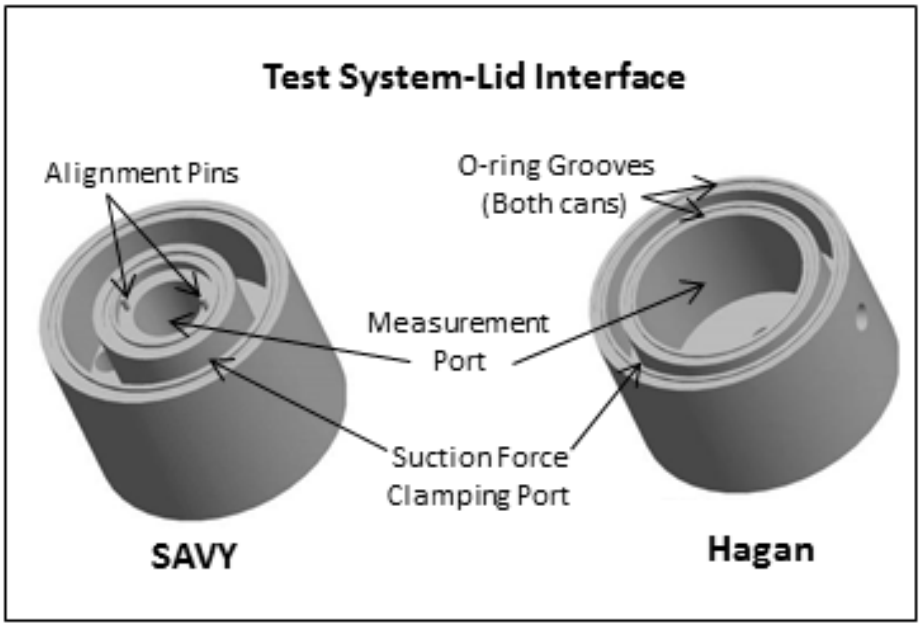


Figure 5. Customized canister interfaces (Title) for the SAVY and Hagan storage canisters.

Figure 6. Photo of the canister interface on SAVY (left) and Hagan (right).

Once the canister interface has been placed over the canister filter, and the IPFT test cycle is initiated, a vacuum air flow is applied to the filter (Figure 7). The canister interface is clamped to the canister by suction force applied to an annulus that concentrically surrounds the measurement port.

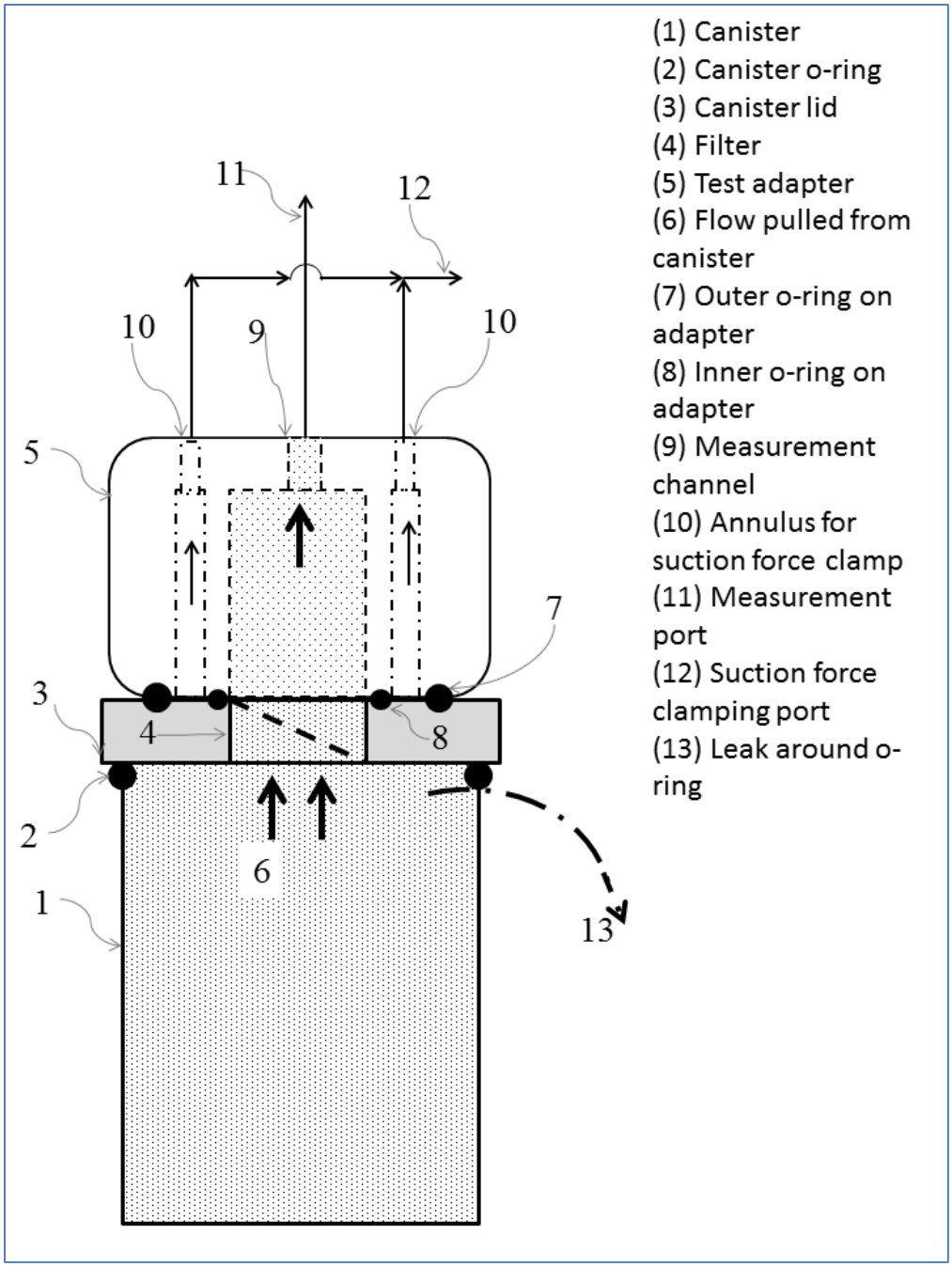


Figure 7. Detail of the internal air flow pathways of the canister and canister interface.

(4) DEVELOPMENTAL EQUIPMENT AND ACTIVITIES

The final Los Alamos IPFT system was developed during an iterative process. Initially, a POC (Proof-of-Concept system) had to be fabricated. This POC device was used to determine the air leakage characteristics of nuclear material storage canisters.

Table 3. The Los Alamos POC (Proof-of-Concept) system (development sequence).

The Los Alamos POC (Proof-of-Concept) system.
(1) The Los Alamos POC system borrowed several capabilities from the cart-mounted Los Alamos FTS (Filter Test System) (Moore ME, Reeves, KP, 2013). The FTS device (Figure 11) measures the leak percent of oil droplet aerosol through a canister filter.
(2) The Los Alamos POC tested and evaluated the canister interfaces (Figures 5 and 6) that connect the SAVY and Hagan canisters to the Los Alamos POC.
(3) Tests performed on actual canisters required an integrated Proof-of-Concept system.
(4) The Los Alamos Proof-of-Concept system quantified the operational air flow rates and pressure drops. This information defined the specifications needed for the customized Isaac® unit that became part of the final Los Alamos IPFT system.

Table 4. The Los Alamos IPFT system incorporates an Isaac® leak tester device.

The Los Alamos IPFT system incorporates an Isaac® device into final design prototype.
(1) Two vendors were approached to build a final Los Alamos IPFT (based on the Los Alamos POC system). The first vendor specializes in mass manufacturing, but they were unsuited to a prototype to execute a design-build process, and they were passed over for the project.
(2) Upon receiving quotes from a second vendor, it was discovered that the cost and complexity of a final IPFT system (based on the Los Alamos POC design) would be higher than purchasing a customized Isaac® leak tester module.
(3) Los Alamos takes advantage of an existing supply chain of leak tester products from the Zaxis company.
(4) Los Alamos did not share knowledge or information to the Zaxis company about the design of the canister interface, the Instrument Development Plan or the Instrument Operating Method.

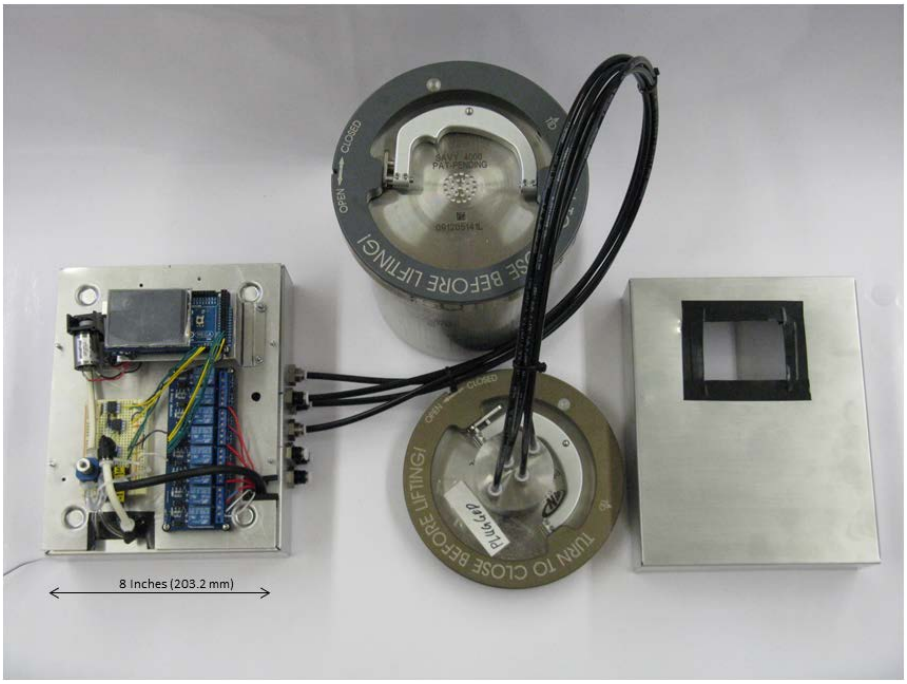


Figure 8. The Los Alamos POC (Proof-of-Concept) system.

Figure 9. The Los Alamos IPFT (In-Place-Filter-Tester) system.

Table 5. The IDP (Instrument Development Plan) bridges the project from start to completion.

Los Alamos POC (Proof-of-Concept) system	**IDP (Instrument Development Plan) – Common Design Tool**	**Final Los Alamos IPFT (In-Place-Filter-Tester) system**
(1) Canister interfaces for Hagan and SAVY canisters		(1) Canister interfaces for Hagan and SAVY containers
(2) Los Alamos POC microprocessor unit and software (see Figure 8)		(2) Isaac® microprocessor unit and software from the Zaxis company
(3) Pumps and valves were integrated into the LANL POC microprocessor unit		(3) A portable, external pump that does not require solenoid controls is utilized.
(4) Instrument Operating Method (IOM) for testing unknown canisters was developed with the LANL POC.		(4) Instrument operating method IOM) for testing unknown canisters was verified with the final LANL IPFT system.
	(5) The IDP (Instrument Development Plan) guided the completion of the POC unit, and was used to verify and measure performance of the final Los Alamos IPFT system.	

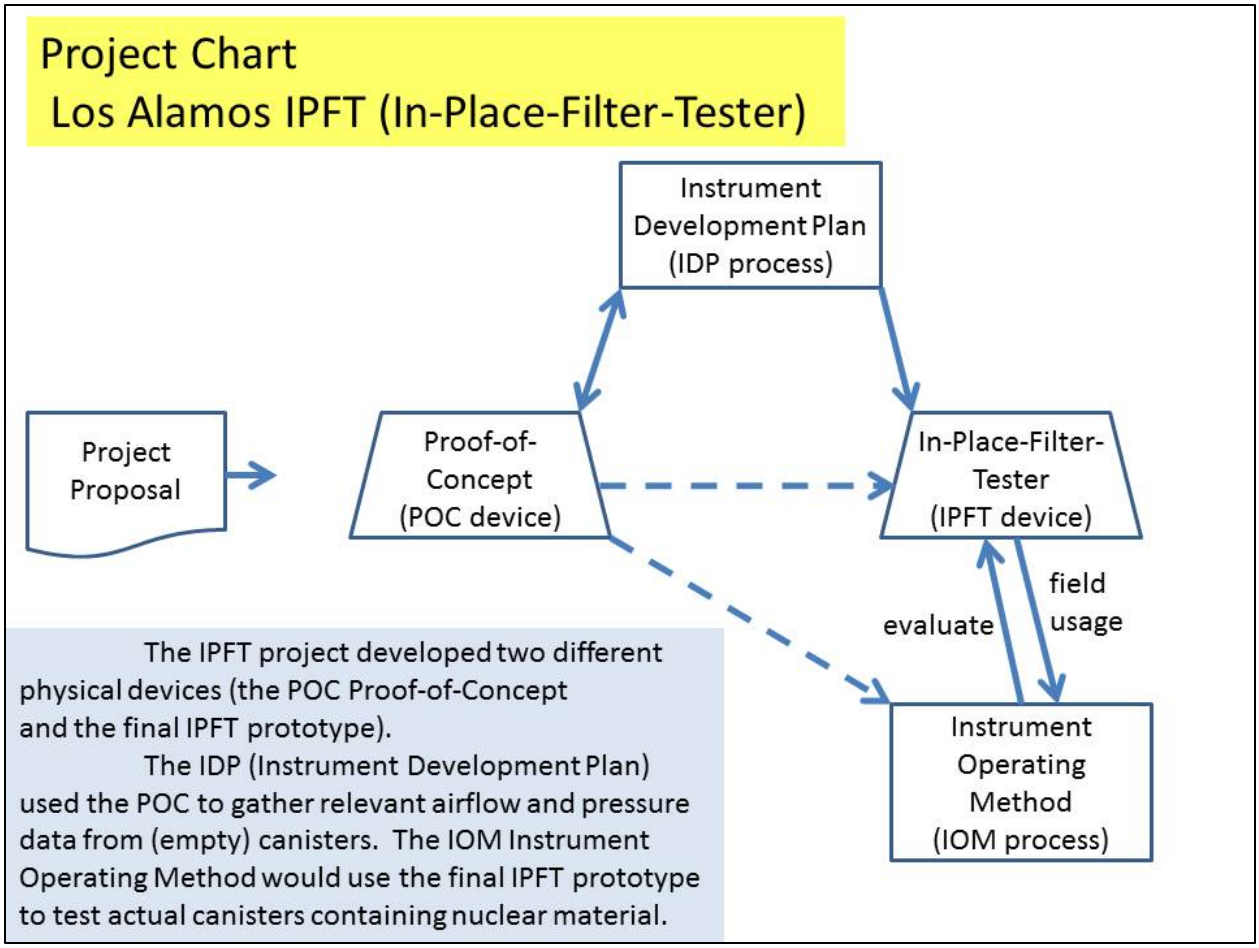


Figure 10. A project chart describing the devices and methods developed in the IPFT project.

Table 6. Comparison of the FTS(Filter Test System) to the IPFT(In-Place-Filter-Tester)

Feature	**Los Alamos FTS (Filter Test System) (Figure 11)**	**Los Alamos IPFT (In-Place-Filter-Tester) (Figure 9)**
Size	4'*2'*4' (L*W*H)	Desktop / portable system
Measures leak of aerosols?	Yes	No
Determines if an assembled can has an air leak?	No	Yes
Requires removal of lid from can?	Yes	No
Contains canister lid clamp (arbor press)?	Yes	No
Determines if filter is clogged?	Yes	Yes
Measures pressure drop through a filter at a specified air flowrate?	Yes	No

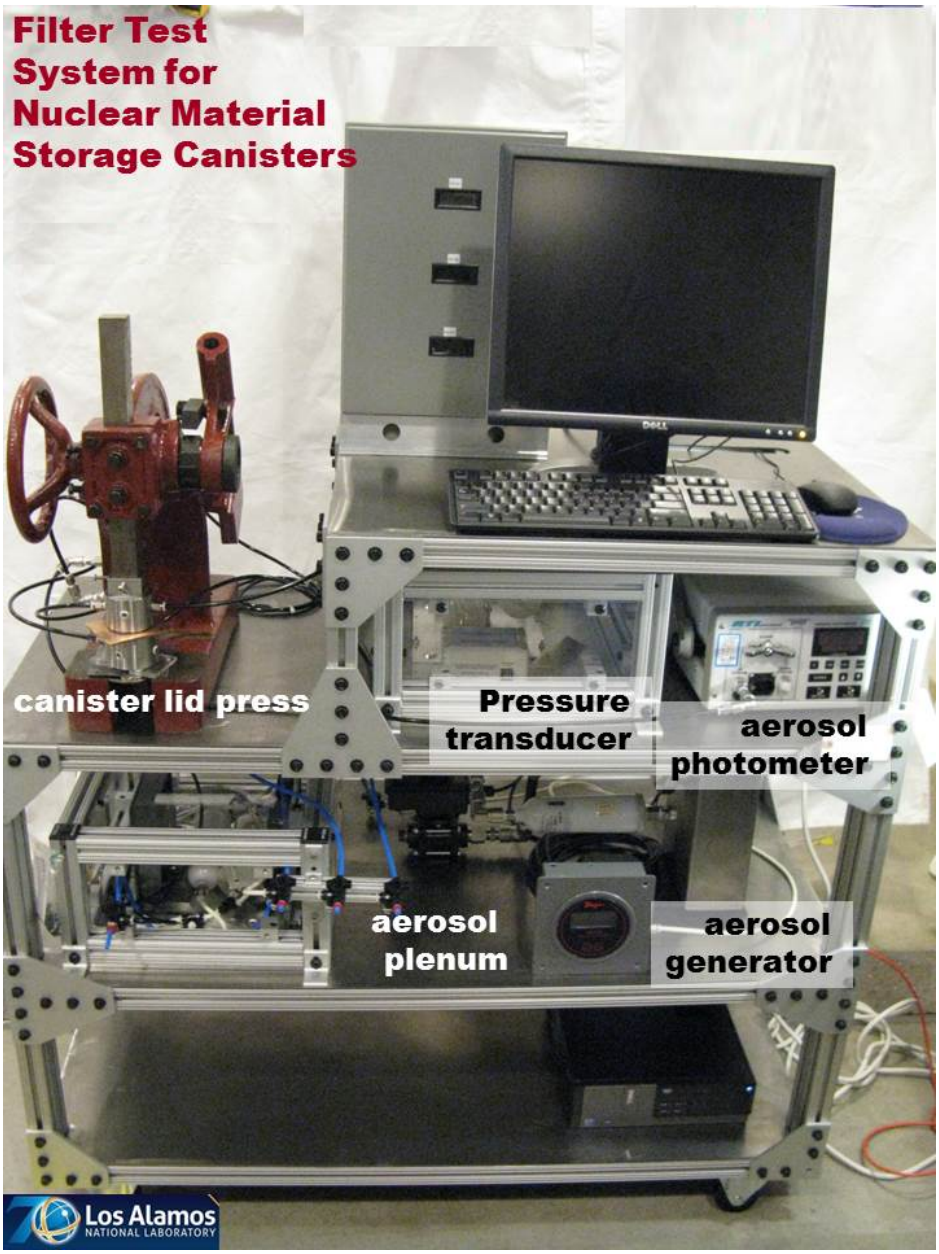


Figure 11. The Los Alamos FTS (Filter Test System) for nuclear material storage canisters. This was developed in 2013, and some of its subsystems were used as templates for the initial Los Alamos POC Proof-of-Concept system.

1. Los Alamos Proof-of-concept (POC)

The Los Alamos POC is a portable desktop unit which encapsulates all pneumatic controls inside its housing. It requires an external power supply to run the pumps, valves and onboard microprocessor unit (an Arduino™ project development board). Aside from the custom fabricated canister interfaces, the remaining electrical and mechanical components are commercially available. The POC software (Appendix I) was created during the Instrument Development Plan process. This system is automatically controlled once an operator initiates the test. The POC device contains independent instrument checks for each testing component, and the capability to self-calibrate its internal working volume. The POC has been described in a complete piping and instrumentation diagram (P&ID) seen in Figure 12. A power distribution and data transfer schematic can be seen in the Build Request (Appendix F).

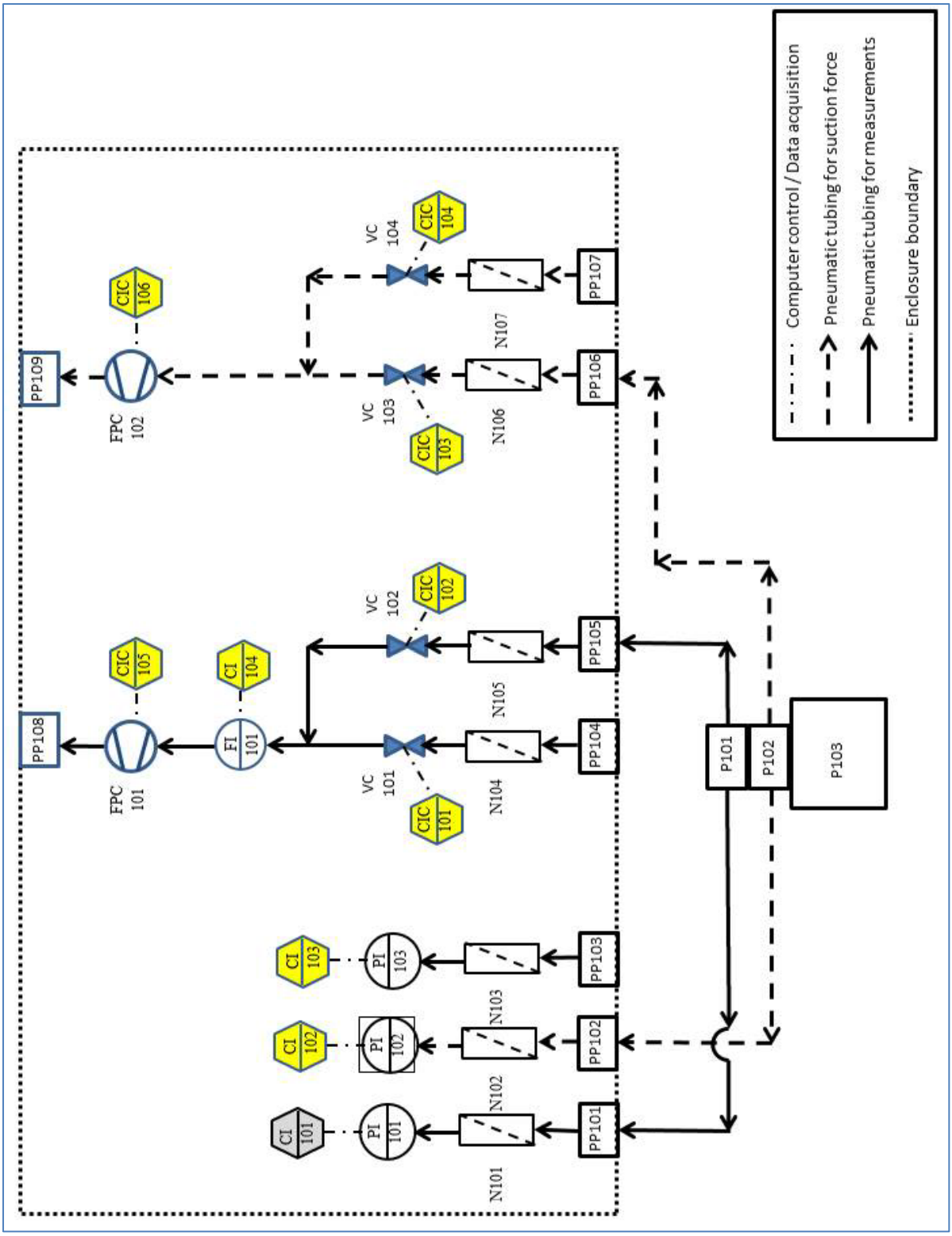


Figure 12. P&ID diagram of the Los Alamos POC (Proof-of-Concept) system. Filters are used for purifying air flow.

Table 7. List of components in the Los Alamos POC (Proof-of-Concept) system.

P&ID Label	**Description**
CI101	Computer indication link to pressure transducer PI101
CI102	Computer indication link to pressure transducer PI102
CI103	Computer indication link to pressure transducer PI103
CI104	Computer indication link to flow meter FI101
CIC101	Computer control link to VC101
CIC102	Computer control link to VC102
CIC103	Computer control link to VC103
CIC104	Computer control link to VC104
CIC105	Computer control link to FPC101
CIC106	Computer control link to FPC102
FI101	Mass flow meter
FPC101	Measurement pump
FPC102	Suction force clamping pump
N101- N107	Filter, HEPA
P101	The canister interface port for measuring canister pressure.
P102	Canister interface port for the suction clamping seal.
P103	The nuclear material canister to be tested.
PI101	Differential pressure gauge (measurement port)
PI102	Differential pressure gauge (clamping port)
PI103	Absolute pressure gauge
PP101-PP109	Bulkhead for vacuum / pressure relief connections
VC101-VC104	Two-way solenoid valve

To start the test process, the canister interface is placed atop the canister lid by the operator. The operator then initiates a command instruction on the main startup screen of the POC software. At this point, the system opens solenoid valve VC103. The computer then activates the FPC102 pump, drawing a vacuum on the suction fitting while at the same time monitoring the vacuum pressure through the PI102 pressure gauge. When the suction fitting has been sufficiently evacuated (current target vacuum is about 185 inWC), the computer closes the VC103 valve and turns off the FPC102 pump. The computer monitors the pressure change (if any) through the PI101 pressure gauge. If a pressure change is detected, the computer will alert the operator to the fact that there is a leak in the suction seal for the system canister interface. (That is, at this point, the canister interface is now sealed to the canister lid, and test measurements may now proceed.)

Once the canister interface is sealed, the computer will begin the canister evaluation measurement. The computer will close the VC101 valve, while opening the VC102 valve. The computer then activates the FPC101 measurement pump, drawing air through the P101 canister interface. While the pump is active, the computer will monitor internal vacuum pressure of the system through PI101 gauge, and mass flow rate from the canister is measured through the FI101 mass flow meter. When a defined (programmable) pressure differential is detected by the PI101 pressure gauge (default test pressure is 5.0 inWC), the system will close the VC102 valve and turn off the FPC101 measurement port pump. The system will then record the pressure rise (if any) across the PI101 gauge for a given time period. This rate pressure change is the defined canister leak rate. The POC Arduino™ computer then stops the test process, opening the VC101 and VC104 valves, allowing ambient air to fill the system. At this point, the test is complete.

The POC computer determines the status of the canister filter (i.e. is it clogged) by first calculating the total volume of the system, determined by a derivation from the ideal gas equation of state:

$$V_{canister} = \frac{P_0 \Delta m}{\rho \Delta P} - V_{instrument} \tag{1}$$

where $V_{canister}$ is the volume of the canister, $V_{instrument}$ is the internal volume of the system between canister interface boundary at measurement port P102 and the measurement pump FPC101, P_0 is the initial atmospheric pressure (measured by the PI103 gauge), ρ is the density of air at current conditions, Δm is the mass of air removed from the system, and ΔP is the final pressure drop across the system (measured by the PI101 gauge). With this setup, Δm may be calculated by equation (2):

$$\Delta m = \int_{t_{start}}^{t_{end}} \dot{m} dt \tag{2}$$

where t_{start} is the start time, t_{end} is the end time, and $\dot{m}$ is the mass flow rate measured by the FI101 mass flow meter. The system will calculate this integral numerically using the trapezoidal rule or another similar numerical integration scheme. The internal volume of the system, $V_{instrument}$, can be determined through a one-time measurement by plugging the inlet at the P102 measurement port and running the system as if it were doing a (normal) canister test. The density of air may be calculated using ambient atmospheric pressure, and assuming a constant temperature and humidity. (These assumptions are likely valid given the expected climate-controlled operating environment of the instrument. However, it may

become necessary to add instruments which measure the temperature and humidity of the air in order to make a more precise calculation of density.)

Once the volume of the system is computed, the onboard computer can analyze the data. If the volume of the canister is calculated to be zero or very near zero, the system will conclude that the filter is clogged, since little or no air passed through the filter during the test. On the other hand, if the volume computed is some amount greater than zero, the system will conclude that the filter is not clogged. Finally, if the volume of the system is computed to be very large (near infinity), then the system will conclude that the canister has a leaky o-ring seal.

The possibility of a leaky o-ring is determined by monitoring the subsequent pressure rise after the sample has been taken. The mass leak rate through the o-ring is given by the following equation:

$$Q_{LEAK} = \left(\frac{V}{P_o}\right)\left(\frac{\Delta P}{\Delta \mathrm{t}}\right) \qquad (3)$$

ΔP is the change measured by the PI101 pressure gauge over the test period (time), and V is the volume of both the instrument and the canister. Upon completing the test, the system will display the status of the canister (e.g. clogged filter, working filter, leaky o-ring, working o-ring).

2. Instrument Development Plan (IDP)

The Los Alamos POC instrument was used to create an Instrument Development Plan:
(1) to measure the air flow and pressure characteristics associated with canister filter clogging, and
(2) to test simulated configurations that mimicked canister leakage paths.

The canister leakage scenarios included quantifying:
(A) air leakage due to foreign material (i.e. dust and hair) fouling of o-rings,
(B) leakage through simulated cracks in o-rings, and
(C) air leakage due to inadequately tightened canister lids.

All of this information must be gathered by experimental measurements, which required an operating Proof-of-Concept (POC) device.

The POC system can determine if a canister filter is clogged, but it must first measure the internal volume of the test system itself. The POC has a calibration feature which measures the system volume when the canister interface is placed on a smooth blank surface (e.g. the blank open space on top of a Hagan container, or a smooth tabletop surface). To establish a criterion, tests were conducted on a 5 quart Hagan canister and its respective internal Vollrath™ canister, as well as on simulated clogged filters.

Table 8 below shows POC measurements for each test. (The estimated volume of dead air space of 2021 cc was estimated by water volume displacement measurements.)

Table 8. Tabular data explained by the filter clogging and canister leak rate flow chart.

Sequential Proof-of-Concept volumetric measurements on a 5 quart Hagan canister		
Volume measure with canister interface on blank surface (cc)	Volume measure with a simulated clogged filter (cc)	Measure headspace volume of a partially filled 5 QT canister.
3.48	-2.92	2108.48
3.57	-2.00	2135.67
3.52	-2.42	2093.68
3.81	-0.66	2093.75
3.89	-1.63	2104.43

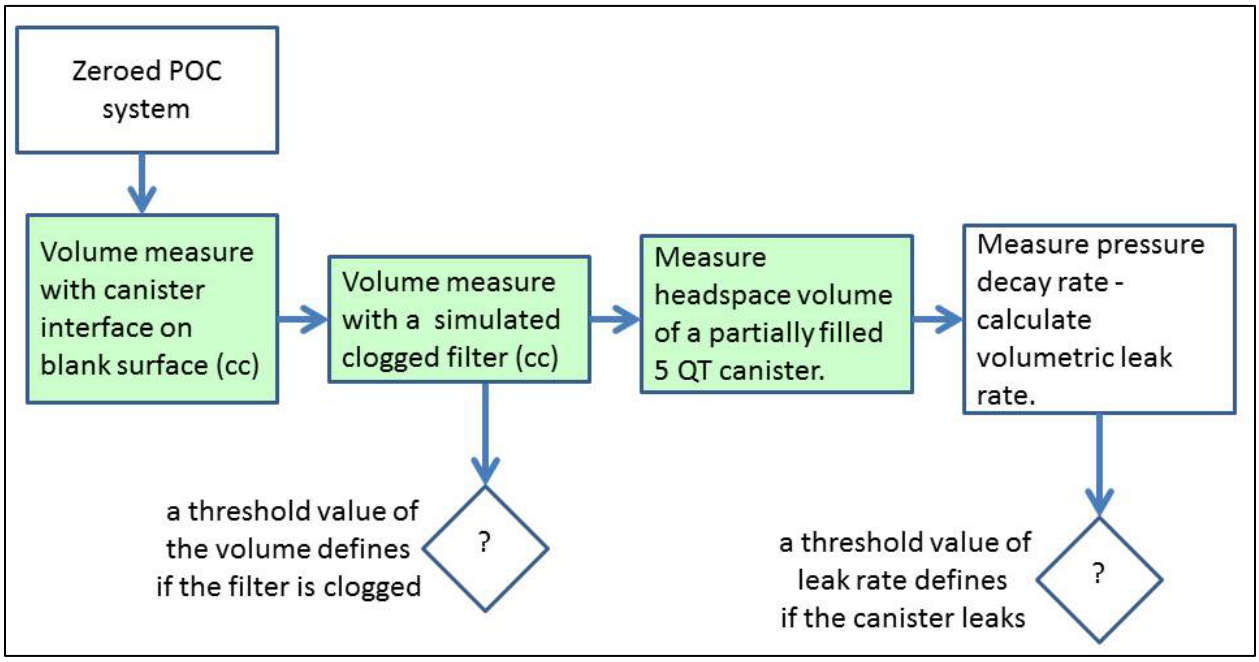


Figure 13. Filter clogging and canister leak rate flow chart.

In Table 8, the sequential volumetric measurements for both the blank surface, and clogged filter tests deviate by less than 1.0 cc, while the larger volumetric test data deviate by roughly 15cc. To further analyze the system's ability to produce accurate measurements, a 5 quart Hagan canister was

incrementally filled with 500ml of water. The POC system was then used to calculate the internal air volume. Figure 14 shows a plot of the actual volume compared with that measured by the system.

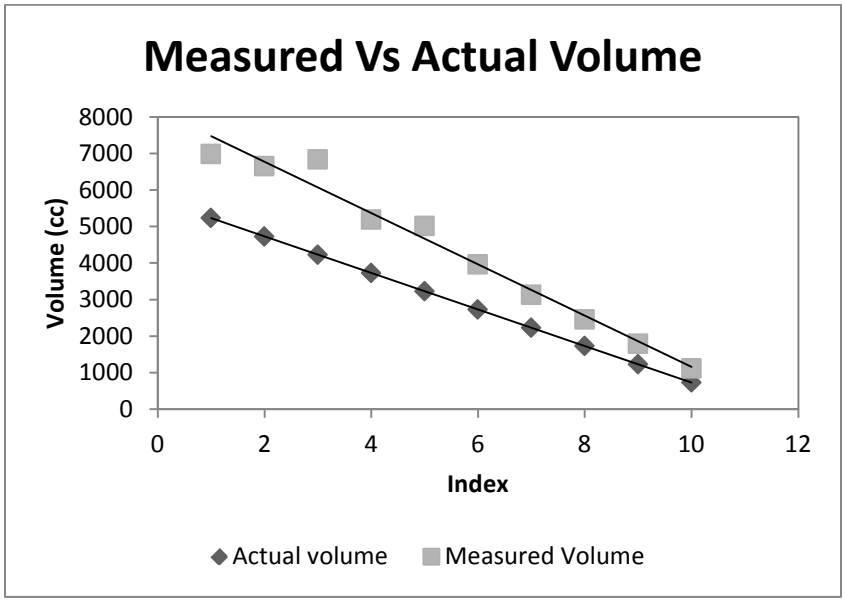


Figure 14. Incremental volumes of water were added to a canister, and POC measurements were compared to the known headspace.

The data (Figure 14) shows a smaller absolute error as the test volume decreases. Because tests on larger volumes require larger air flow volumes, and longer times to achieve test pressure requirements, the data acquisition system incorporated into the POC experiences an increase in signal drift. Any inherent leaks in the system or canister integrity will also affect the calculated volume, by definition, based on the mass flow of air during the test. This data was used to produce a preliminary volume criterion to distinguish between a clogged filter, and a fully packed canister. The POC declares any test volume within 100cc of the calibrated system volume as a clogged filter.

To correctly tighten the Hagan canister lid, the lids have a reference for the sealed position of the lid (Figure 15). After roughly seven complete revolutions, the line on the lid should be flush with the midpoint of the sealed reference frame (the solid black box). To establish a convention for lid location, a “negative 1.0 inch” to “positive 1.0 inch” coordinate system is also depicted in the figure.

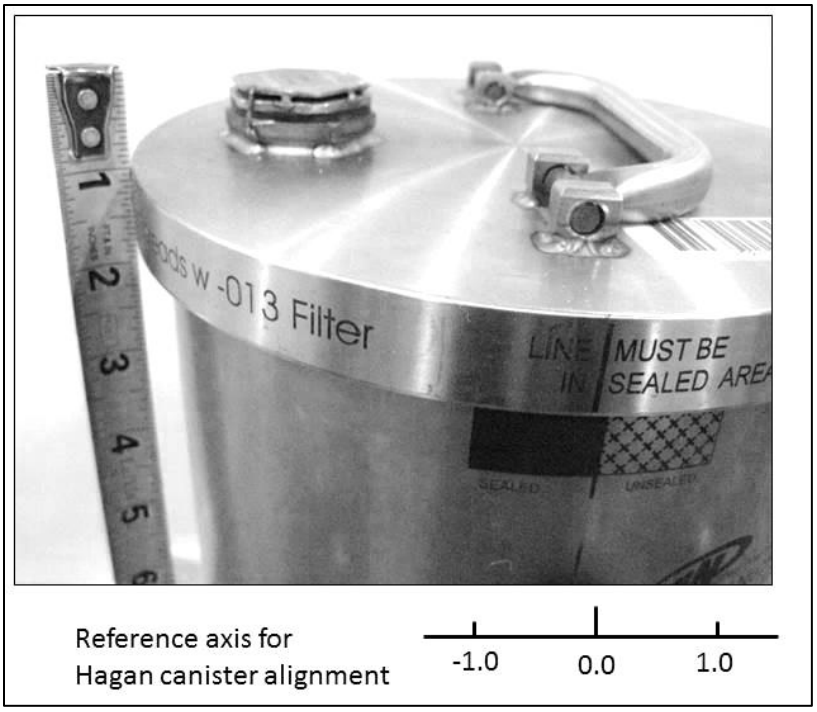


Figure 15. Hagan canister alignment marks.

Early tests showed inherent leak rates in the POC Proof-of-Concept system (with no attached canister or canister interface) to be about 0.004 cc/sec. Hagan lids installed to the sealed alignment mark with un-fouled o-rings measured a leak rate at about 0.1 to 0.2 cc/sec. "Arizona road dust" (developed as a test contaminant for air-filter testing) was used for these measurements. Commercially available road dust of about 9µm diameter (ISO 12103-1, Powder Technology Inc, Burnsville, MN) was used as an intentional fouling agent on Hagan canister o-rings. The o-rings were coated with a thin layer of silicon grease, and then covered with a layer of road dust. Leak rates averaging $0.13\,\frac{cc}{sec}$ resulted after this intentional fouling, but the leak rate was within the measured range of un-fouled o-rings. As such, the POC system could not identify a leak caused by this fouling agent.

O-ring polymers can exhibit cracks due to ageing and degradation, especially after exposure to elevated temperatures, and after radiation exposure. Cracks were simulated by partially cutting halfway through the thickness of the o-rings. The accumulation of such effects degrade the ability of the o-ring to form a leak tight seal. Figure 16 shows an actual aged o-ring with visible cracks, and Figure 17 indicates the coordinate system used to describe the direction of the cuts that were made halfway through the diameter of tested o-rings.

Figure 16. Actual aged o-ring

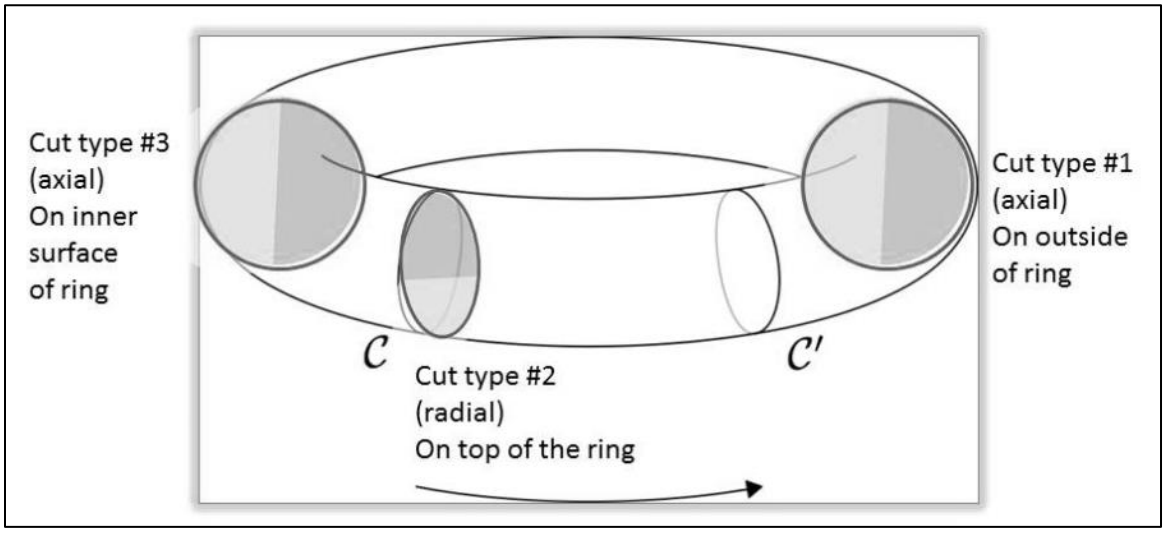


Figure 17. O-ring coordinate system for cutting (slicing) the o-rings.

The leak-tight integrity of the Hagan lid was measured with multiple simulated cracks. Leak rates were measured from the "-1.0" alignment mark (completely tightened, see Figure 15), and measurements were

continued in ¼ inch increments as the lid was loosened. This test shows the effect of simulated cracks (type #2, Figure 17) on the leak rate, as a function of lid location.

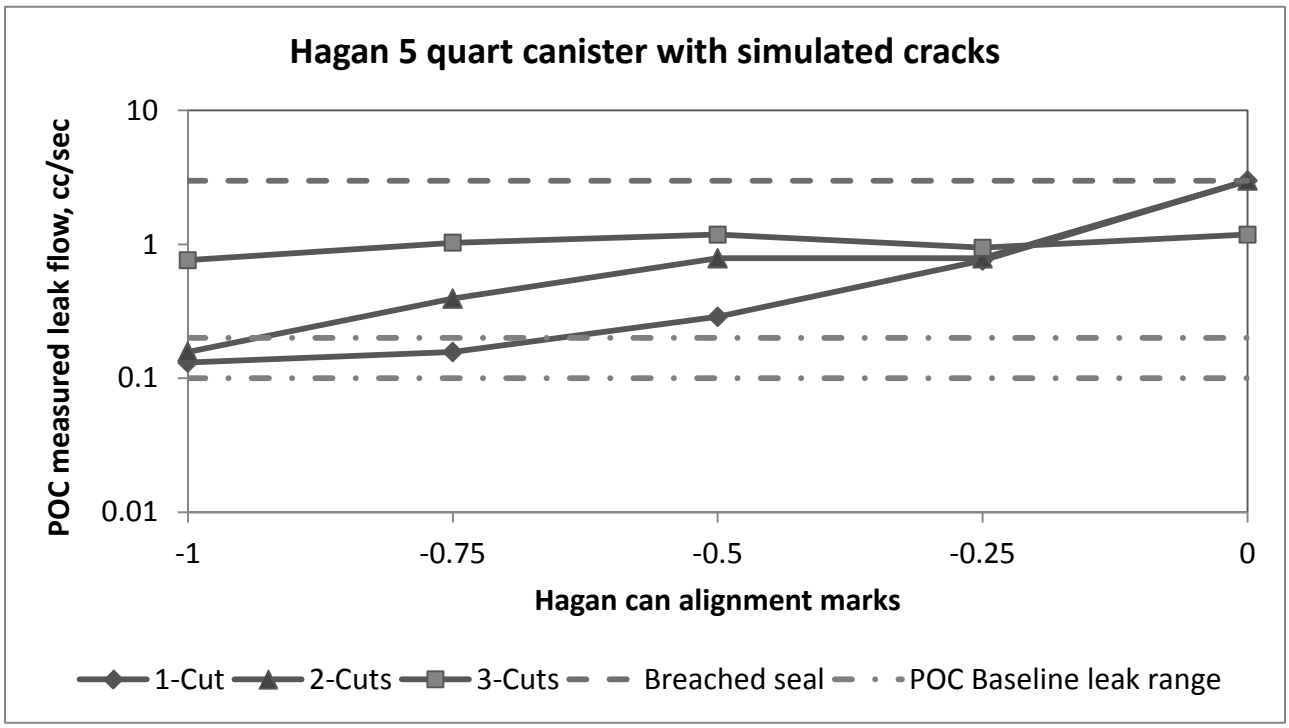


Figure 18. Simulated degradation test performed on a Hagan 5 quart canister. This test covers multiple sealed locations.

The leak rate increases as the number of cuts increases from one, to two, to three (Figure 18). While the leak rate for one or two cuts remains within the range of a sealed canisters measurements, both cases with one or two cuts in the o-ring fail to hold vacuum at the "0.0" alignment mark (Figure 18). Results using the final IPFT system (see next section) are shown in Figure 19.

Both Hagan and SAVY canister designs require an active user input (i.e. a force manually applied by hand) to compress the o-ring and "activate" the o-ring seal. Without a lock pin mechanism (see Table 2) present in the Hagan design, the lid and thus the o-ring seal might become loosened over time. Further tests were conducted over the entire canister alignment range to identify leak characteristics of an inadequately tightened lid. These tests (see Figure 20) did not include any intentional fouling agents or simulated degradation (i.e. cuts in the o-ring) effects.

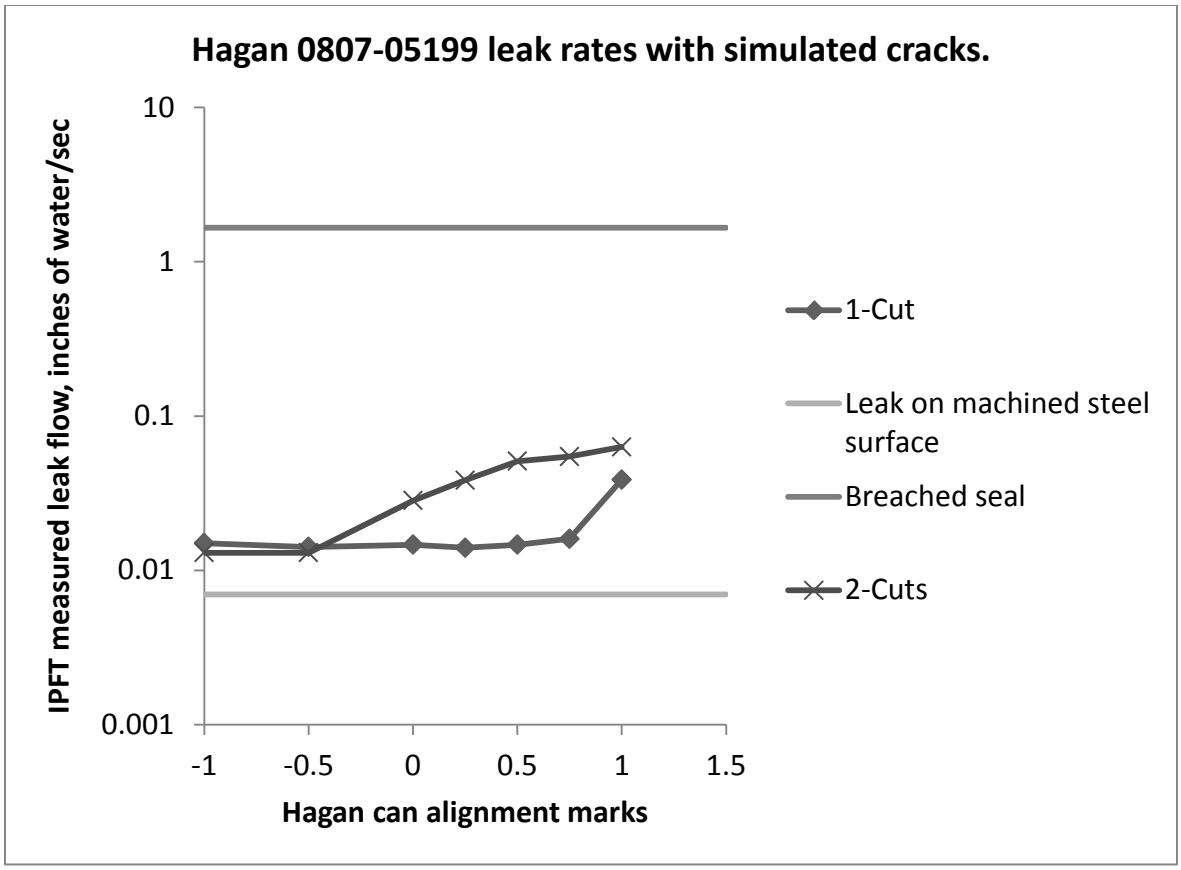


Figure 19. ISAAC-Hagan - 0807-05199_O-ring 03 pg 148.xlsx

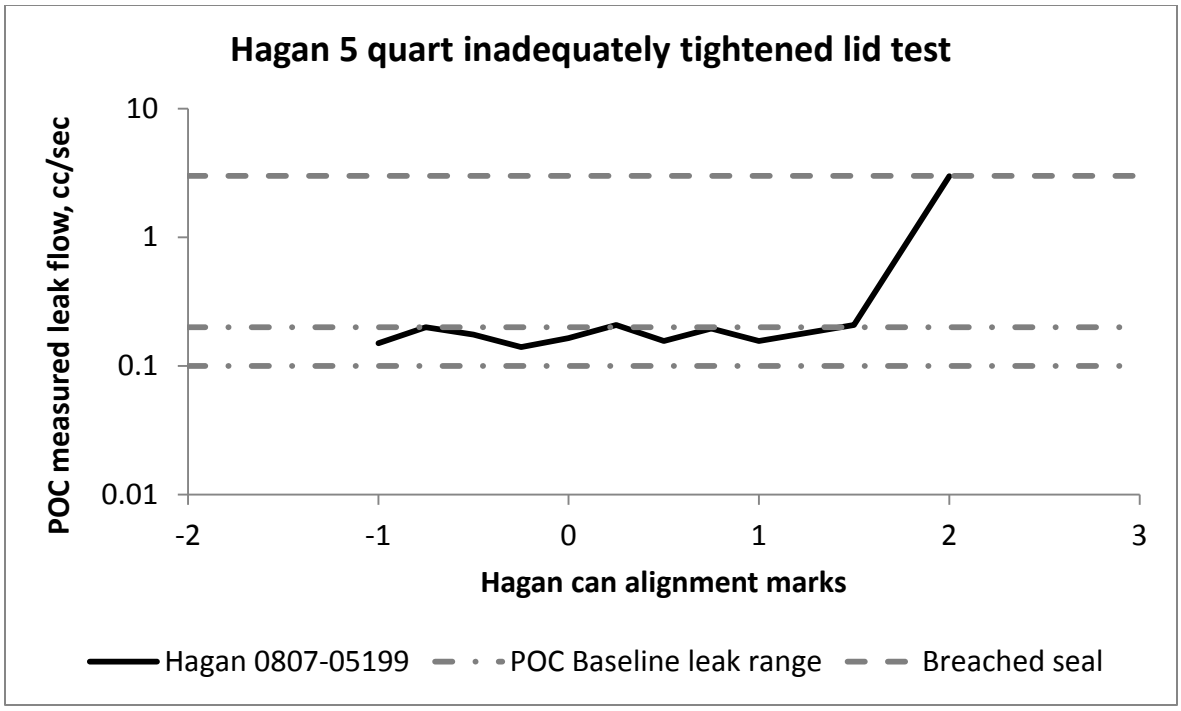


Figure 20. Incremental loosening of the lid with associated leak rates (with unaltered o-ring).

In Figure 20, if the lid is loosened up to a full (1.0) inch past the sealed alignment mark, the canister leak rate does not deviate from previous baseline measurements of leak rates of unaltered canisters. When the Hagan canister lid was moved to the "+2.0" inch alignment mark, a breached seal resulted (where the canister cannot even maintain a test vacuum).

In a similar manner, removal of the internal o-ring completely prevented the achievement of an internal test vacuum, implying that the POC device could identify a leak caused by a missing o-ring.

3. Incorporation of the Isaac ® unit into the LANL IPFT

Data from the IDP Instrument Development Plan was used to specify parameters for a customized Isaac™ leak tester from the Zaxis Company. The Isaac™ leak tester system is the user interface for the LANL IPFT In-Place-Filter-Tester, and provides a pathway for future canister test activities. Table 8 compares the first POC Proof-of-Concept system to the final LANL IPFT device (which includes the Isaac leak tester).

Table 8: POC Proof-of-Concept compared to IPFT In-Place-Filter-Tester.

Test / feature / issue	**POC capability**	**IPFT capability**	**Is this a LANL/DOE request or requirement?**
Filter clog test.	Yes	Yes	Yes
Canister leak measure	Yes	Yes	Yes
Measure volume of dead air space	Yes	Possible-TBD	No
Volume leak rate (cc/min)	Yes	Possible-TBD	Yes
Pressure decay rate (inWC/sec)	Yes	Yes	N/A
Canister interfaces provide a measurement port and also suction clamping for canister evaluations.	Two canister interfaces were developed for Hagan and SAVY canisters	The same canister interfaces were utilized in the final IPFT prototype.	N/A
Knowledge of the operational parameters of flowrate and pressure decay (from the IDP Instrument Development Plan)	The parameters were developed and measured by the LANL POC (Proof-of –Concept) device	These parameters were needed to specify the custom Isaac unit, which was incorporated into the LANL IPFT final prototype.	N/A

The Los Alamos IPFT system utilizes the canister interfaces in the same manner described for the POC Proof-of-Concept device. After following the Leak Test Procedure (Moore ME, 2013A) the canister interface still would not provide a perfect seal, and an intrinsic leak rate was measured as a threshold value in the IPFT system. The lower IPFT measurement range was measured (Figures 21 and 22) as the leak rate of the Isaac™ instrument with a plugged sample measurement port. Tests with multiple

canisters, fouling agents, and simulated crack degradation tests define the IPFT measurement range. The upper boundary of the IPFT measurement range is defined by the largest measured leak rate measured by the experiments during the IDP (Instrument Development Plan) process.

Current Los Alamos leak test procedures for the Hagan and SAVY canisters specify a helium gas leak test on clean, empty storage canisters. Based on mass spectrometry, the helium leak test is about three orders of magnitude more sensitive than the air leakage method described in this project report. The lower boundary of the helium leak range is the SAVY-4000 leak criterion and the upper helium range is the SAVY-4000 post-drop test helium leak criterion (Anderson LL et al 2013).

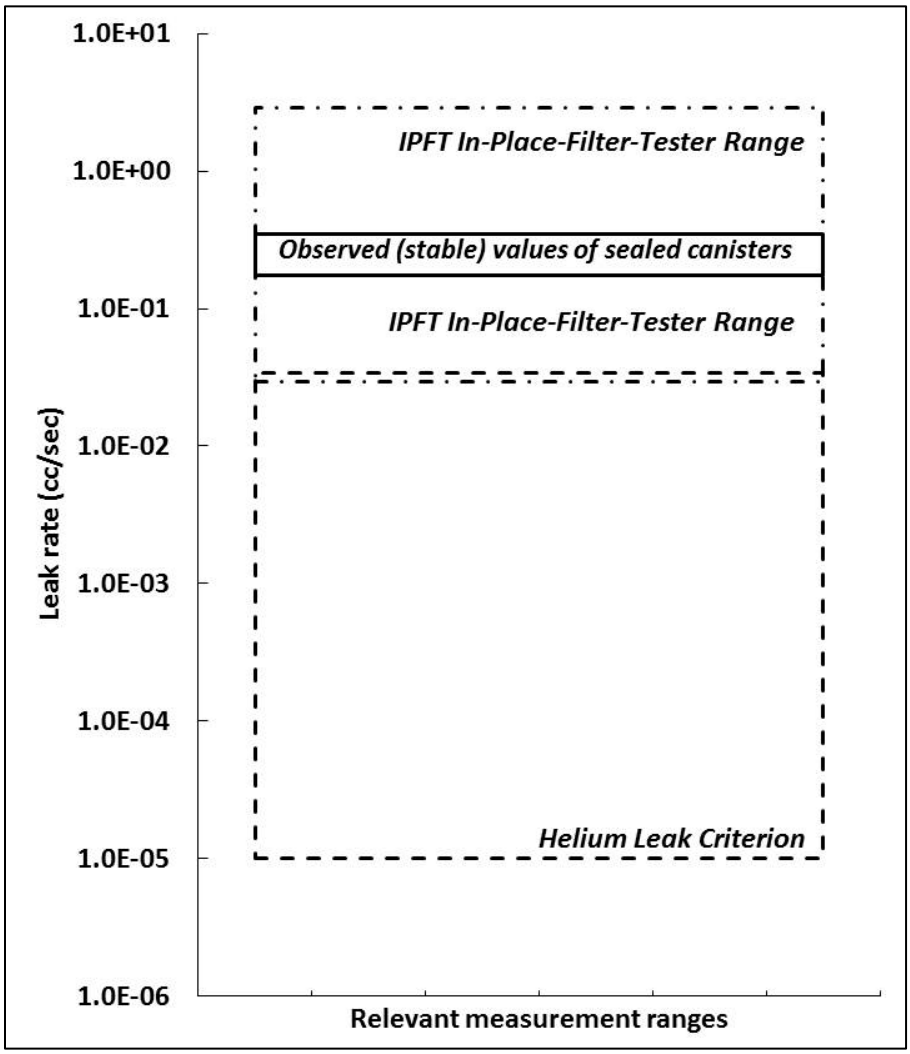


Figure 21. Relation between helium leak rates and the LANL IPFT In-Place-Filter-Tester test range.

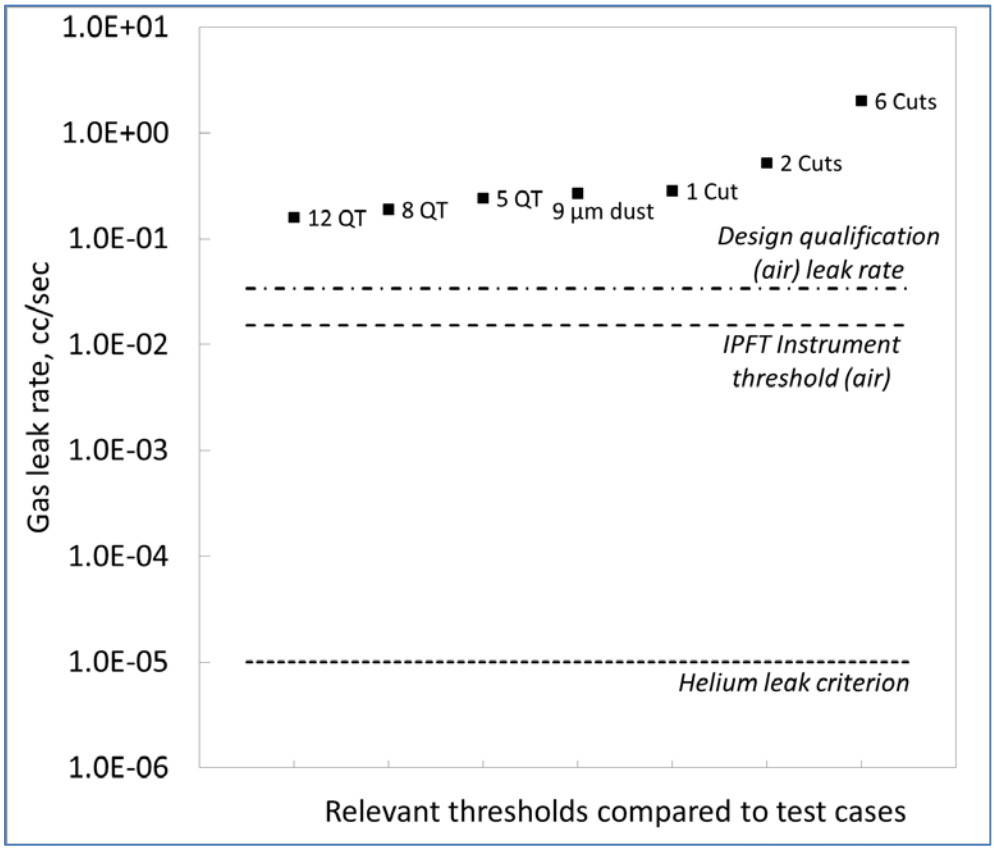


Figure 22. Relevant thresholds compared to measurements of IDP (Instrument Development Plan) tests.

(5) DELIVERABLES

1. Los Alamos In-Place-Filter-Tester (IPFT)

The IPFT system does not currently have a flow meter to measure volumetric air flow rate, therefore, the presence of a clogged filter is determined on the time interval required to attain a specified vacuum inside the tested canister. During application of vacuum to the tested canister, the pump air flow rate decreases as the pressure differential increases. The canister pressure eventually plateaus at the test value, and air flow then stops. Smaller volumes experience a more rapid pressure change due to the restriction of gases available for removal. Figure 23 shows a characteristic pressure rise over time as the pump evacuates the test volume (fill section) of an empty five quart canister, and simulated clogged filter.

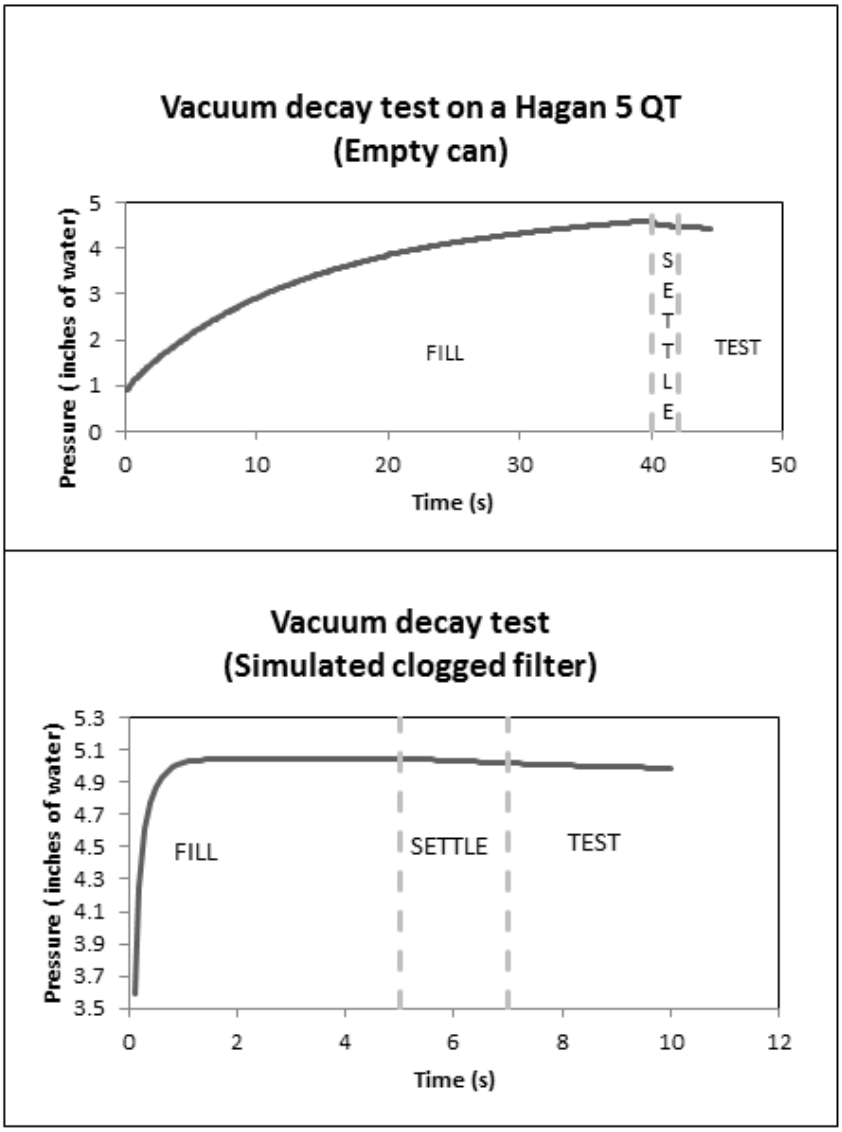


Figure 23. Vacuum decay test comparison - Hagan functioning vs clogged filter

The empty 5 quart canister, with a volume of 5380 cc, exhibits a more gradual curve over the 40 seconds needed for the test to reach the established pressure. The pressure vs time curve seen in the clogged filter simulation shows a much more abrupt transition from ambient to test pressure, taking less than 2 seconds. Canisters in the as-found condition will be storing different volumes of materials, and with working filters will have curves that lie in between the empty can and clogged filter data sets. To establish a minimal fill time (to signify a completely clogged filter) time versus pressure tests were conducted on Hagan 5, 8 and 12 quart containers loaded with their respective internal Vollrath™ slip containers (Table 9).

Table 9: Fill time vs. maximum test pressure achieved for multiple configurations.

Clogged Lid			**Hagan 8 QT 08/06-08083** **Vollrath 88060**		
Fill Time (S)	Max Test P (in H2O)	Within Tolerance ? (Yes / No)	Fill Time (S)	Max Test P (in H2O)	Within Tolerance ? (Yes / No)
2.2	4.93	Yes	45	4.9	Yes
2.1	4.91	Yes	43	4.89	No
2	4.87	No	40	4.87	No
Hagan 5 QT 08/07-05199 **Vollrath 88030**			**Hagan 12 QT 09/02-12213** **Vollrath 88080**		
Fill Time (S)	Max Test P (in H2O)	Within Tolerance ? (Yes / No)	Fill Time (S)	Max Test P (in H2O)	Within Tolerance ? (Yes / No)
35	4.9	Yes	85	4.92	Yes
31	4.89	No	75	4.84	No
28	4.85	No	60	4.65	No
25	4.8	No			
20	4.64	No	45	4.34	No
15	4.4	No			

The empty volumes remaining in each canister that mimic as-found conditions are large enough to require, at a minimum, 35 seconds for the IPFT to reach the necessary test volume. This data was combined with clogged filter simulations to establish a minimum fill time test capable of identifying a clogged filter.

The first POC Proof-of-Concept system established the flow and pressure parameters used in the sampling and clamping portions of experimental canister testing. Based on early data sets, a customizable leak test system (the Isaac™ from Zaxis) was specified and purchased. The stainless steel canister interfaces are interchangeable between both the first POC device and the final IPFT system. The proven programming and user interface incorporated from the Isaac™ leak test module provides a high confidence of repeatability for scenarios involving multiple testing apparatus and/or technicians. The LANL IPFT system runs with external pumps supplying both the measurement and clamping vacuum. The clamping flow passes directly from the canister interface to its supplying pump, while the flow which is used for all measurement purposes is routed through the inner workings of the Isaac module. A dial

gauge on the back of the test unit regulates the flow and pressure supplied from the external pump to the measurement port.

The Isaac™ software allows user input to control the specified test pressures and tolerances, the acceptable leak rate, and the amount of time for each section of the test being performed. The sections of a standard leak test include: "fill", "settle" and "test" (Figure 24). During the fill stage a valve opens the measurement port to the established measurement vacuum. If the test pressure is within the specified tolerances at the end of the programmed "fill" time, the Isaac system will close the valve to the pump and start the "settle" stage. During this stage, a steady state leak is achieved. The "test" stage allows the Isaac system to monitor the change of pressure (inWC) over the specified test time interval.

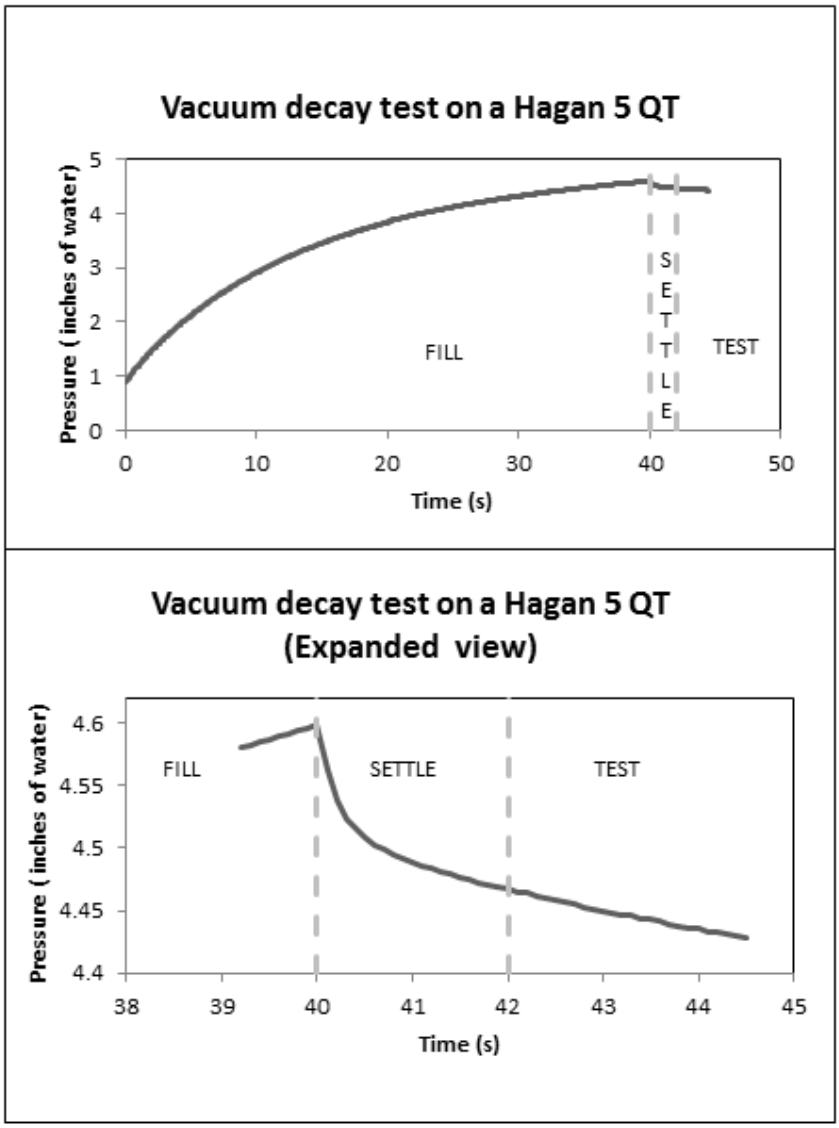


Figure 24. Vacuum decay test including expanded view

2. Instrument Operating Method (IOM)

The IOM (Instrument Operating Method) requires user programing and decision making to properly assess as-found canisters that contain nuclear material. With both pumps and the Isaac™ unit turned on, the canister interface is placed atop the canister lid. The vacuum air flow in the outer annulus of the canister interface immediately begins to apply suction force to clamp the canister interface to the canister lid. In the lower right hand corner of the touchscreen interface, a numerical value with units of inches of water (inWC) displays the vacuum exerted from the clamping flow on the internal volume of the can. A decrease in this pressure is indicative of a leak between the concentric clamping and measurement ports. This will happen during setup, but must equalize and maintain a constant value to insure that the interface is sealed correctly. The manual valve located in the measurement port of the canister interface is opened to reset the canister volume to ambient pressure conditions, and closed once again for tests. If the internal pressure of the canister exceeds 0.2 inWC vacuum, or surpasses a pressure decay rate of 0.01 inWC/sec, it is assumed that the inner o-ring of the canister interface is "leaking vacuum" (i.e. the canister interface shall be reseated according to the IOM).

Once the canister interface is sealed, the minimal fill time test parameters are entered into the Isaac settings menu according to Table 10, and the test sequence is started (see IPFT flowchart in Figure 25). If the filter is clogged, the leak test system experiences a smaller volume than if the filter was not clogged, and the test pressure is achieved within a minimal fill time. If the test pressure is not reached, a new fill time is prescribed based on container size (Table 10). The fill time is now sufficient for an unclogged filter. If the test pressure is still unable to be reached with the new fill time, canister leak-tight integrity cannot be confirmed, and the Isaac tester displays a low pressure warning. Thus, either the o-ring fails to meet the leak rate criteria or the canister itself is leaking. Once the proper test pressure has been achieved, the leak-tight integrity of the container can be quantified. Leaks within the defined range will be shown in the lower right hand corner of the touchscreen to one ten thousandth of an inch of water column (0.0001 inWC). Any leak rate larger than that specified during the test setup will immediately trigger the Isaac unit to stop testing, and notify the user that a gross leak was detected. It should be noted that the leak rate tolerance can be set to the same value as the test pressure to produce numerical leak rate values covering the largest range. In this configuration it will be left to the user's discretion to identify leak rates which are not tolerable, the Isaac unit will declare a passing leak rate for every test.

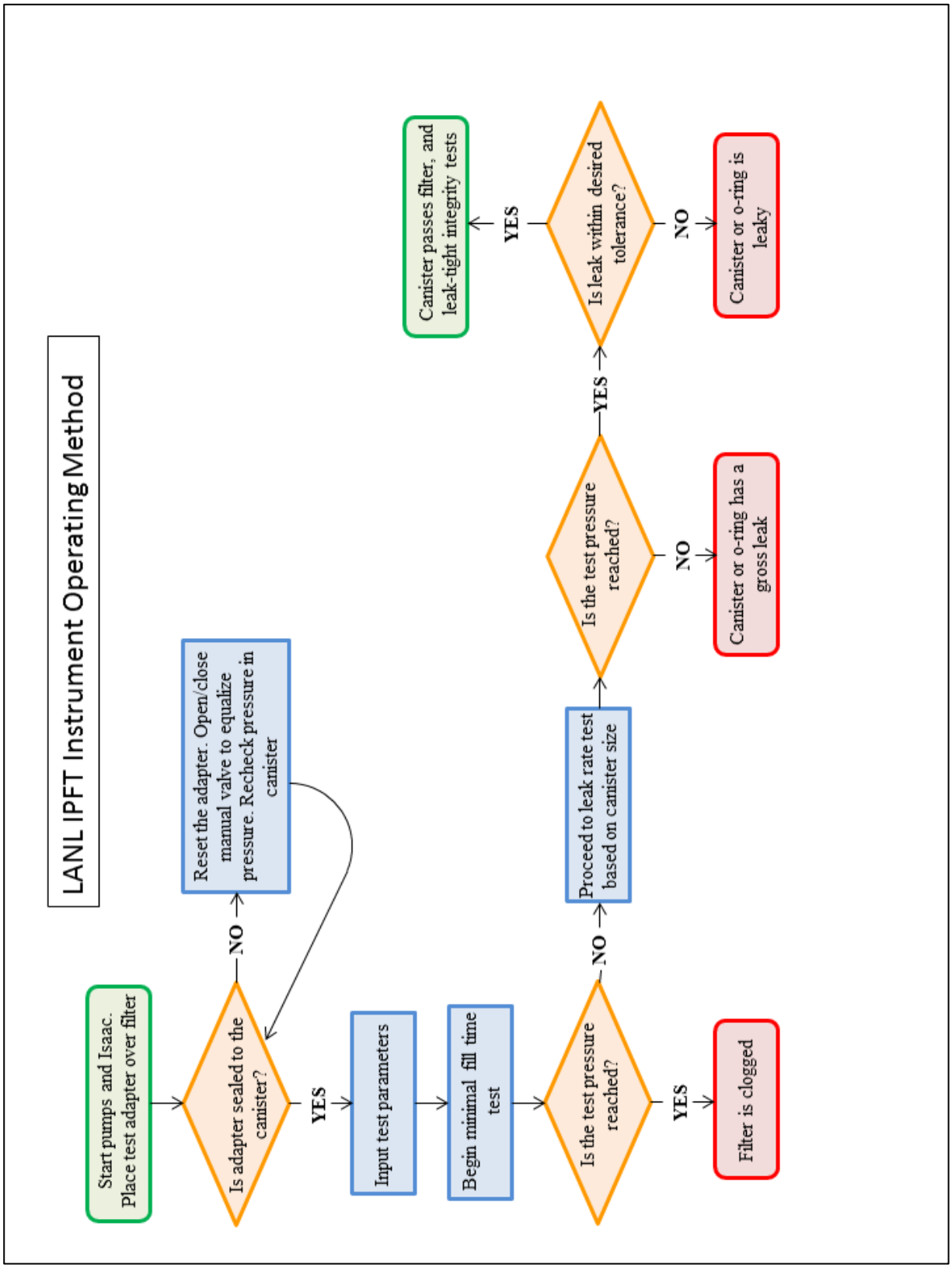


Figure 25. Instrument Operating Method flow chart

Table 10. Established test parameters to be applied to the IOM

Test Parameters		
Test pressure		5.00 in WC ±0.1
Actual supplied pressure		5.00 in WC ±0.05
Settle Time		2 Seconds
Test Time		3 Seconds
Fill Time		Variable
Leak (Vacuum decay) tolerance		5 in WC
Minimal fill time		
Fill time for both Hagan and SAVY		2.5 Seconds
Test times for multiple canister sizes		
Nominal size (QT)	Actual Size with tubing and canister interface included(cc)	Fill time
5 (empty)	5380	40 Seconds
8 (empty)	8865	70 Seconds
12 (empty)	12827	130 Seconds
5, with Vollrath 88030	2021	35 Seconds
8, with Vollrath 88060	3038	45 Seconds
12, with Vollrath 88080	4427	85 Seconds

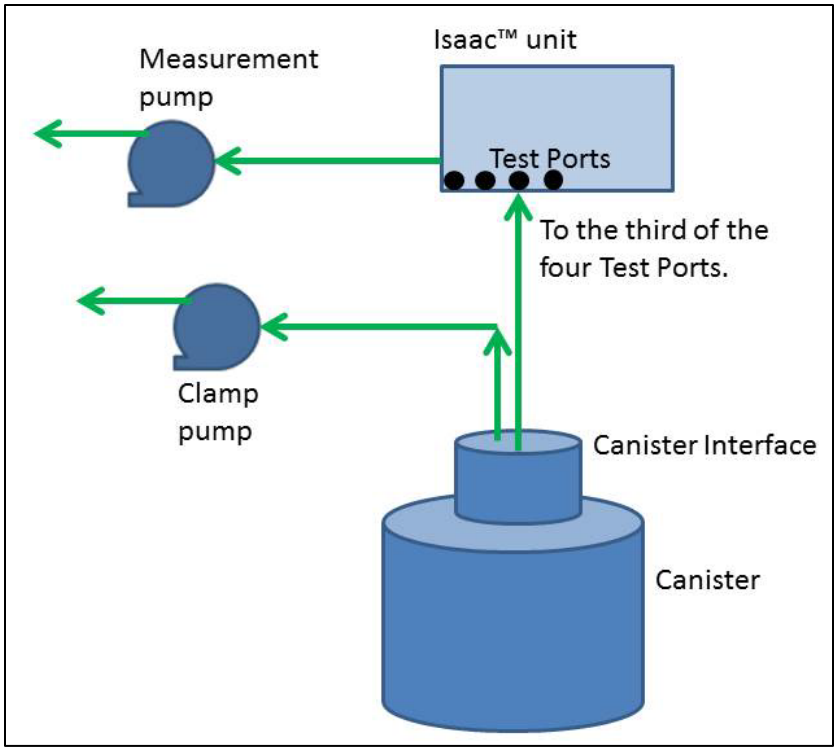


Figure 26. IPFT In-Place-Filter-tester system assembly

Isaac HD

Font Panel

1. **Touch Screen Interface (TSi)** - The liquid crystal touch screen display is the primary control of the leak tester.
2. **Abort Button / Fail Indicator Light / Retest Button** - Red LED indicator marked with an '**X**'.
3. **Start Button/Pass Indicator Light** - Green LED indicator marked with a '**√**'.
4. **Test Ports** - Each port is a 1/8"-27 F NPT (similar to R1/8 BSPT British Standard Pipe Taper) female thread. This is where the DUT (device under test) is attached. Test port positioning will vary by model.
5. **Tilt-up feet** – The position of the tester can be raised by rotating the tilt-up feet into position. Units with the Electronic Regulator option will have these feet removed due to the position sensitivity of the electronic regulator.

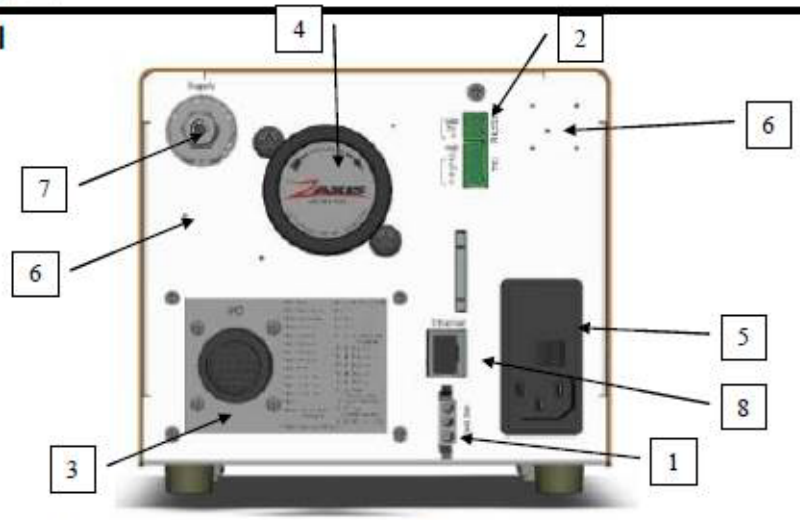
ZAXIS

6

Figure 27. The third port from the left of the Test Ports (#4) is connected to the Canister Interface Measurement Port.

Isaac HD

Rear Panel



1. **Foot Switch Connector:** For starting a test. Any closed contact switch may be used to start a test by shorting pins 1 & 3

   Mating Conn. & Pins – Tyco Electronics 1-480304-0 (Digi-Key A1400-ND), 60618-1 (Digi-Key A1422-ND)

2. **Serial Output:** Connection for interfacing Isaac to computers or printers. (Wiring)
3. **Option Space:** This space is used for Clamping (pneumatic outputs) or I/O connection (shown)((Wiring))

   If the I/O space is filled clamps will be in the secondary air connection location.

4. **Pressure Regulator:** Controls the air pressure applied to the product under test.
5. **Power Entry Module:** This component includes the power cord socket, on/off switch, and the fuse holder. Power entry is capable of a range of 109-255~VAC. 2A Fuse (Service instructions)
6. **Secondary Air Connections:** These positions have multiple uses; Flow control for ramping burst/crack pressure, coupling or pilot air entry, or secondary test air supply. See the Quick Start shipped with the tester for details.
7. **Supply:** (Air or other test gas) Connection is 1/8"-27 F NPT (Similar to R1/8 BSPT British Standard Pipe Thread) Supply gas must be clean and dry.
8. **Ethernet Connector:** ASCII communication, data output and command string input.

Additional items may be installed on the rear panel, see the quick start guide shipped with the tester for additional details.

ZAXIS

7

Figure 28. The Supply Port (#7) is connected to the Measurement Pump.

(6) SUMMARY

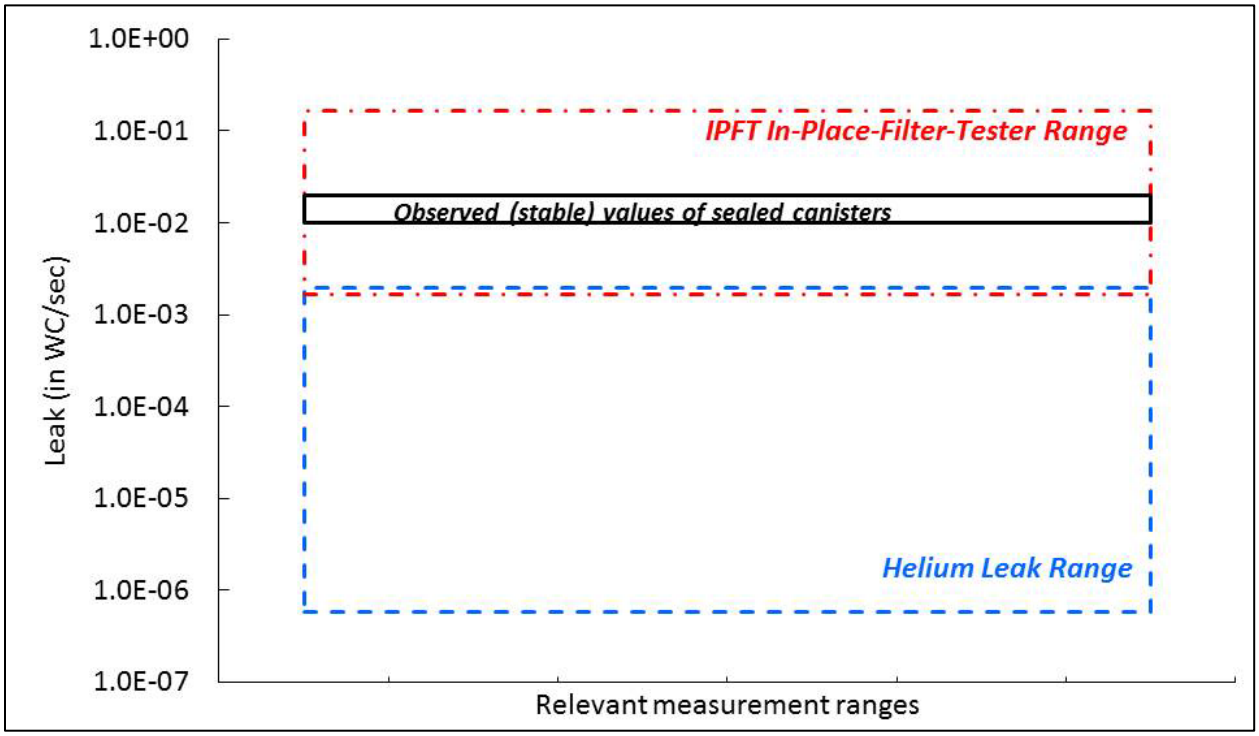


Figure 29. Relation between helium leak rates and the LANL IPFT In-Place-Filter-Tester test range.

Both the POC (Proof-of-Concept) and the Final IPFT (In-Place-Filter-Tester) systems assess the filter and o-ring functionality on nuclear material storage canisters, without requiring removal of the canister lid. An IDP (Instrument Development Plan) provided preliminary results which served as pertinent information to the construction of the final IPFT. An IOM (Instrument Operating Method) was then developed as a means to systematically test canisters for filter clogging, fouled o-rings, and overall leak-tight integrity. The need for such a device comes from the demand of containment capabilities involved with nuclear material storage. The very nature of nuclear storage combines radioactive contamination with a gas flow being driven by the pressure force of hydrogen generated inside the containers and by pressure gradients due to atmospheric air pressure fluctuations. Diagnosing the condition of container filters and canister integrity is important to ensure worker and public safety and for determining the handling requirements of legacy apparatus.

Hagan and SAVY-4000 canisters are employed in Los Alamos National Laboratory storage. The newer SAVY construction canisters are designed to have a maximum normal operating pressure range of

0.25 to 0.50 inWC (1 to 2 kPa), and a filtration capability of capturing greater than 99.97% of 0.45 micron mean diameter aerosol. Sufficiently thin walls are implemented for content inspection, thus overpressure can result in visible deformation. The lid-canister interface is sealed by compression of an o-ring, thus non leak-tight conditions would present a pathway for radioactive material contamination. Current designs have a “no-maintenance” life of five years, however, the desired life extension to a service life of 40 years or greater would substantially reduce laboratory resources. As o-ring degradation and filter lifetime tests increase canister service life, the necessity for in-service canister surveillance will increase to meet design criteria and detect potential problems. The current evaluations include visual deformation inspections for canister overpressure caused by clogged filters, and contamination surveys indicating possible o-ring seal failures.

Two canister interfaces were designed to provide a connection for the different styles of canister lids, and the testing hardware. Each canister interface is designed to supply a clamping vacuum (with an external pump) to create a seal with the canister lid. An additional measurement port is routed through both the leak test system and an internal channel of the canister interface before it penetrates through the filter for test data.

The POC and IPFT systems collect and analyze data in different manners due to their varied mechanical constructions. The Los Alamos POC instrument was used to create an Instrument Development Plan: (1) to determine the air flow and pressure characteristics associated with canister filter clogging, and (2) to test simulated configurations that mimicked canister leakage paths. The canister leakage scenarios included quantifying: (A) air leakage due to foreign material (i.e. dust and hair) fouling of o-rings, (B) leakage through simulated cracks in o-rings, and (C) air leakage due to inadequately tightened canister lids. The POC determines if a filter is clogged and it measures leak rate in terms of volumetric flow rate, since it has an air flow meter. With less than a 0.7% deviation from a calibrated system volume, clogged filter simulations were confidently identified. Un-fouled o-ring configurations were used to establish stable canisters at a leak rate between 0.1 to 0.2 cc/sec. This baseline leak range was then used to characterize the tests in the Instrument Development Plan. Leaks from both simulated cracks, and inadequately tightened canister lids were distinguishable from un-fouled, properly sealed canisters. The Los Alamos POC instrument determined pertinent air flow and pressure quantities which were used to specify a customized Isaac® (Zaxis, Salt Lake City, UT) leak tester device. The Isaac® module was then incorporated as the user interface for the final IPFT.

Tests from the Instrument Development Plan validated the IPFT system for determining clogged filters, and for the canister leak measurements based on pressure decay rates. To establish a minimal fill time, which upon being reached would signify a completely clogged filter, time versus pressure tests were conducted on Hagan 5, 8 and 12 quart containers. The results were used to create test parameters and an

associated minimum fill time which could distinguish clogged filters from fully functional ones. When compared with current helium gas leak test procedures, the IPFT is about three orders of magnitude less sensitive. It was necessary to develop an observed leak rate range of stable canisters to characterize leaky conditions. As with the POC, leaks from both simulated cracks, and inadequately tightened canister lids were observable. These results indicate that the Los Alamos IPFT instrument is capable of determining if a filter is clogged and the leak rate conditions in nuclear canister, without requiring removal of the container lid.

The IPFT determines if a filter is clogged, and it measures leak rate in terms of the pressure decay rate. The IOM (Instrument Operating Method) systematically walks a user through the required sequence to properly assess as-found canisters. Test parameters are prescribed based on the type of test (clogged filter or leak rate condition) as well as the size of canister. Full compliance to the IOM produces all the necessary results for as-found canister analysis.

The IPFT measures pressure decay rate from 1.7E-03 inWC/sec to 1.7E-01 inWC/sec. On the same unit scale, helium leak testing of canisters at Los Alamos has a range from 5.7E-07 inWC/sec to 1.9E-03 inWC/sec. For a 5-quart storage canister, the IPFT measures equivalent leak flow rates from 0.03 to 3.0 cc/sec. The helium leak volumetric flow rate criterion is 1.0E-05 cc/sec, and the upper range of the helium leak range is 0.034 cc/sec. (This is the design qualification leak rate, which is the criterion for post-drop tested canisters). The IPFT does not provide the same sensitivity as helium leak testing, but is able to gauge the assembled condition of as-found and in-situ canisters.

PART II.CANISTER FILTER TEST STANDARDS FOR AEROSOL LEAK RATES

Filter Standards Protocol: Execution, Data Gathering, Evaluation and Conclusions

A series of filter test standards were needed to verify performance of the Los Alamos FTS (Filter Test System, Figure 11) for nuclear material storage canisters. The LANL FTS measures (1) the percent leak of oil droplet aerosols through a canister filter, and (2) the pressure drop across a filter. Both of these tests are performed at air flows of 200 cc/min, and both tests require the canister lid to be removed from the canister for the experimental tests.

Verify the Los Alamos FTS matches the NFT results – using older Hagan NCR canister lids

The LANL FTS was completed before the IPFT In-Place-Filter-Tester project, and conversations about the FTS operation were instrumental for pursuing the possibility of determining the "as-found" condition of unopened storage canisters.

When the NFT company manufactures canister lids, they also measure the aerosol leak rate and pressure drop for those canister lids. Since October 2002, they have used a set of fourteen Hagan canister lids as a library of "standard filters" (Figure 30). These lids have never been placed "in service" with nuclear material and have been marked as "NCR" (Non Compliance Report). The NCR lids were sent from NFT to Los Alamos for round-robin comparison testing of the two filter test systems.

Figure 30. NCR Hagan lids used for system comparison.

Testing at Los Alamos of the Hagan canister NCR lids commenced in June 2013, and the first set of tests at Los Alamos did not match the NFT historical data set. Based on statistical (t-test) analysis, the Los Alamos aerosol leak and pressure drop measurements deviated (unsatisfactorily) from the NFT results. A series of equipment and procedure modifications were executed to reconcile the test results from Los Alamos and NFT (Golden CO).

LANL FTS equipment modifications pertinent to measuring aerosol leak rate

- Initially, a TSI model 3079 aerosol generator was being used, but it was removed and replaced by a modified ATI model 6D aerosol generator. (The TSI 3079 delivered the correct air flow rate for the FTS system, but it produces an aerosol distribution larger than the SAVY-4000 test requirements.)
- The ATI model 6D aerosol generator was modified to use a smaller pump, in order to scale the air flow rate to the 200 cc/min requirement of the canister tests. (The ATI 6D is designed to produce aerosol for air flows of 1000 CFM (cubic feet per minute), therefore, the original factory pump was oversized for this application.

LANL FTS equipment and process modifications pertinent to measuring pressure drop across the filter

- Operation of the electronic flow controller was examined, and was set to the correct reference pressure. This issue was explained and rectified. (Moore ME 2013b).
- The LabView software driver was periodically registering null (zero) values for pressure drop data in its data acquisition string. This issue was corrected.
- The altitude density correction. The pressure drop measurements (across the tested filters at Los Alamos) were corrected to the altitude at Golden CO, where the NFT company performs their measurements. The values calculated as "$\Delta P_{LANL\text{-}corrected}$" were then directly compared to the values measured at Golden CO as "ΔP_{NFT}".

$$\Delta P_{LANL\text{-}corrected} = \Delta P_{LANL\text{-}measured} * (R_{Golden}/R_{LosAlamos}), \quad (4)$$

where

R_{Golden} = the ratio of atmospheric pressure at Golden CO to sea level pressure (i.e. 0.81), and,

$R_{LosAlamos}$ = the ratio of atmospheric pressure at Los Alamos to sea level pressure (i.e. 0.76).

After these system modifications were performed, the fourteen Hagan NCR canister lids were re-measured for aerosol leak rate and for filter pressure drop (Figures 31 and 32). The statistical comparison of both data sets is presented in Table 9 (Milton JS and JC Arnold 1986).

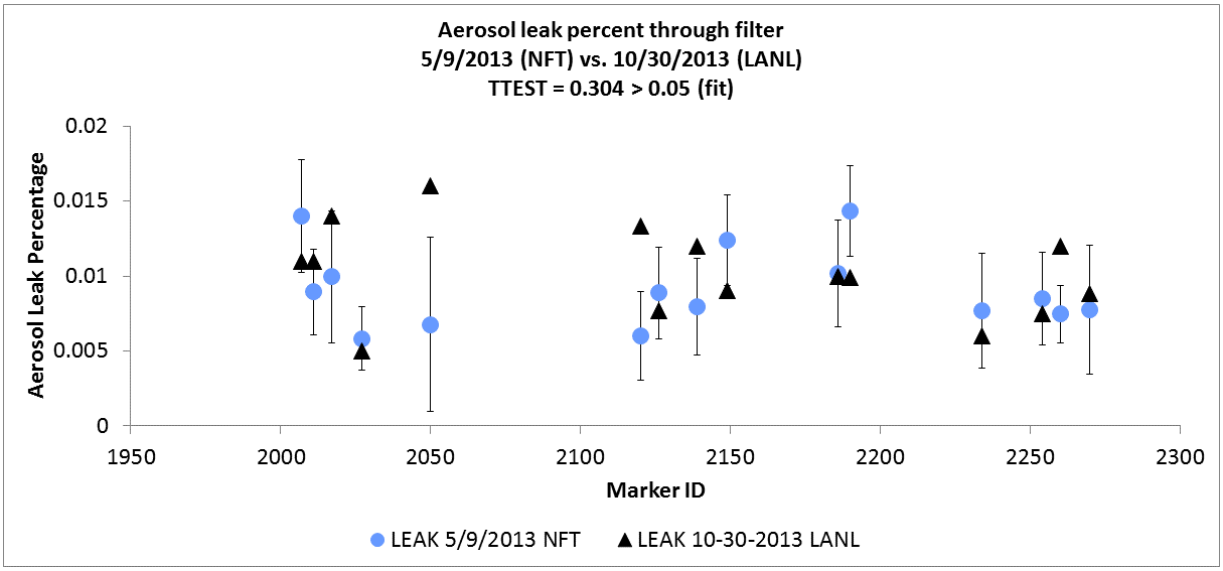


Figure 31. Los Alamos leak percent tests (on Hagan NCR lids) compared to NFT results.

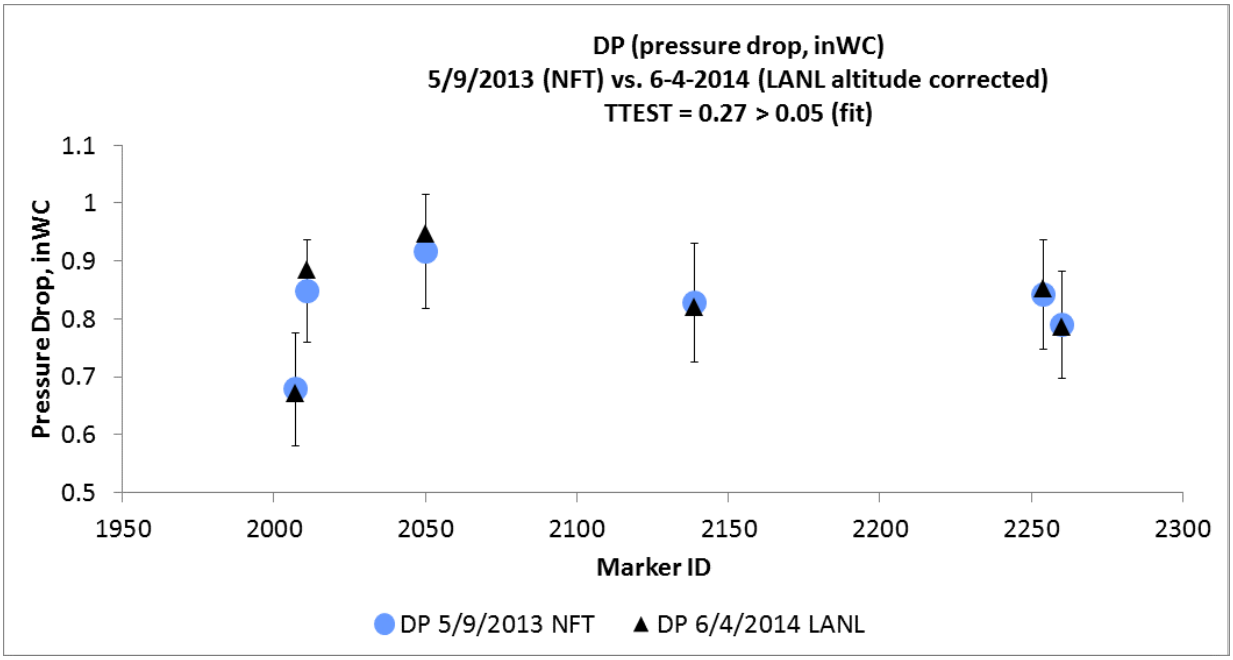


Figure 32. Los Alamos pressure drop tests (on Hagan NCR lids) compared to NFT results.

Table 9. Paired t-test summary compares the LANL results to the NFT data.

Paired t-test (two tails)	**p-value**	**Answer to the mean test**
Are the measured differences in aerosol leak (in between NFT and LANL) due to chance? *(PEN% 5/9/2013 (NFT data) versus PEN% 10/30/2013 (LANL data))*	0.30	**Yes.** (The p-value is 0.30 > 0.05) Conclusion: the data sets are a statistical match.
Are the measured differences in pressure drop (in between NFT and LANL) due to chance? *(DP 5/9/2013 (NFT data) versus DP 06/04/2014 (LANL data))*	0.27	**Yes.** (The p-value is 0.27 > 0.05) Conclusion: the data sets are a statistical match.

Procurement of new SAVY-4000 style filter standards

A series of reference standard filters based on the SAVY-4000 filter design were purchased (Figures 33 and 34). These filter standards will be used to verify the performance of Los Alamos FTS Filter Test System located in the Los Alamos plutonium facility (a rad area) in relation to the FTS Filter Test System located in the Aerosol Engineering Facility (a non-rad area). There were fifteen filter standards procured. Three of the fifteen filter standards were manufactured with two layers of the ceramic filter material while the other twelve were manufactured with the standard three layers of material. These filters came with a test performance sheet and will provide another metric for comparing the Los Alamos systems to the manufacturer's system. On the SAVY-4000 filters, a leak (percent) of aerosol through the filter is not detectable in most cases (Table 10), due to a lower limit of measurability of the photometer used to measure the aerosol concentration. The filter standards with two layers of filter material were intended to provide a measured leak percentage larger than 0.0001%. The data sheets supplied with the filter standards indicate that a percent leak is still at 0.0001% while the pressure drop across the filter did decrease. A sample of these filter standards will be intentionally damaged with a "pinhole leak" so that a measurable amount of aerosol leakage may be measured.

Figure 33. Top surfaces of the Hagan, SAVY-4000 and NFT filter standard (clockwise from upper left).

Figure 34. Lower surfaces of the Hagan, SAVY-4000 and NFT filter standard (clockwise from upper left).

Table 10. Performance summary of the new SAVY-style standard filters.

Filter ID	Pressure Drop	% Leak
243568-2L-001	0.64	0.0001
243568-2L-002	0.66	0.0001
243568-2L-003	0.66	0.0001
243568-3L-001	0.77	0.0001
243568-3L-002	0.76	0.0001
243568-3L-003	0.78	0.0001
243568-3L-004	0.82	0.0001
243568-3L-005	0.73	0.0009
243568-3L-006	0.72	0.0001
243568-3L-007	0.82	0.0001
243568-3L-008	0.74	0.0001
243568-3L-009	0.91	0.0001
243568-3L-010	0.68	0.0001
243568-3L-011	0.82	0.0001
243568-3L-012	0.87	0.0001

Table 10 contains the manufacturer data for each of the lids. The filter ID contains information regarding the number of layers of filter material. As seen in Table 10, the percent leak on almost all the filter standards are 0.0001%.

As seen in the side-by-side photos (Figures 33 and 34), the NFT filter standard is not a complete lid, since it lacks an outer latch mechanism, outer ring and handle. These filter standards can still be utilized with the LANL FTS Filter Test System, since the filter standards will be held in place with the arbor press (Figure 11) located on the FTS.

ACKNOWLEDGEMENTS

This project was funded by the US Department of Energy Health, Safety and Security Program, WAS Project No.: 2013-HS-2013008, B&R No. HU1004500.

REFERENCES

Anderson LL, Blair MW, Hamilton EJ, Kelly EJ, Moore ME, Smith PH, Stone TA, Teague JG, Veirs DK, Weis E and Yarbro TF, Safety Analysis Report for the SAVY 4000 Container Series, Revision 3. Los Alamos National Laboratory, Los Alamos Controlled Publication, LA-CP 13-01502, 2013

ASTM F2338-09. Standard Test Method for Nondestructive Detection of Leaks in Packages by Vacuum Decay Method. American Society for Testing and Materials. 2013.

ASTM E2930-13. Standard Practice for Pressure Decay Leak Test Method. American Society for Testing and Materials. 2013.

Brassell GW and Brugger RP. "Bonded carbon or ceramic fiber composite filter vent for radioactive waste", U.S. Patent Number 4,500,328; 1985.

Havrilla GJ, Bowen S. 2002 LAUR 02-7568 RFETS Drum Filter Evaluation with Stainless Steel Containers - Final Report, Los Alamos National Laboratory Unclassified Report. 2002.

Havrilla GJ 2000 LAUR-00-3034 Characterization of RFETS Drum Vent Filters, Los Alamos National Laboratory Unclassified Report. 2000.

Milton JS and JC Arnold. Probability and Statistics in the Engineering and Computing Sciences. McGraw-Hill. 1986.

Moore ME, Smith PH, Veirs DK, Anderson LL, Klemm R, 2011. Low Flow Filter Efficiency Testing, Los Alamos National Laboratory, Los Alamos Unclassified Report LAUR 11-10891, 2011

Moore ME, Reeves, KP, 2013. Filter Measurement System for Nuclear Material Storage Canisters - End of Year Report FY 2013. Los Alamos National Laboratory, Los Alamos Unclassified Report LAUR-14-20641.

Moore ME, 2013a. RP2 RIC Aerosol Engineering Facility Leak Test Procedure, Los Alamos National Laboratory, RP2-RIC-DP-77, R0.

Moore ME. 2013b. Pressure drop measurement - reconciled - LANL vs. NucFilt filter - flow controller 8-15-13.docx. Personal communication. Los Alamos National Laboratory. 2013.

Stone TA, Smith PH 2011. Los Alamos National Laboratory overview of the SAVY-4000 design: meeting the challenge for worker safety. Los Alamos National Laboratory, Los Alamos Unclassified Report. LAUR-11-03986.

APPENDICES

See separate file.

APPENDIX CONTENTS

A. Product information sheet – Los Alamos IPFT In-Place-Filter-Tester

Los Alamos NATIONAL LABORATORY EST.1943

PRODUCT INFORMATION

The Los Alamos IPFT (In-Place-Filter-Tester) system for nuclear material storage canisters

An In-Place-Filter-Tester device was developed at the Los Alamos National Laboratory to determine the "as found" condition of unopened storage canisters. Customized stainless steel fittings (canister interfaces) were developed to provide a connection between the different styles of canister lids, and the testing hardware.

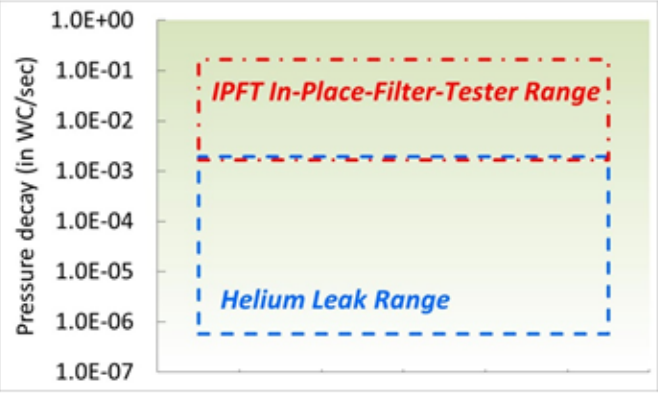
The US Department of Energy uses several thousand canisters for storing nuclear material in different chemical and physical forms. In these canisters, specialized filters allow gases to escape, and maintain an internal ambient pressure while containing radioactive contaminants.

In a storage container, the interface between the lid and the canister is sealed by an o-ring, and a leak at the o-ring would create a pathway for release of radioactive contamination. Monitoring programs can assess o-ring degradation and filter performance to extend canister service life. Therefore, in-service canister surveillance assists in meeting design criteria, measuring current conditions and anticipating future problems.

A portable instrument that assesses the filter functionality and leak-tight integrity of nuclear material storage canisters, without requiring removal of the canister lid

APPLICATIONS

The Los Alamos IPTF In-Place-Filter-Tester is compact, lightweight, and portable. The instrument is capable of determining filter clogging and canister (leak) integrity, without requiring removal of the container lid.



OPERATIONS

The Los Alamos IPFT system includes a custom Isaac® (Zaxis, Salt Lake City, UT) leak tester device. A dedicated Los Alamos procedure (the Instrument Operating Method) measures canister properties:

- Filter functionality (clogging), and,
- Pressure decay rate (o-ring integrity).

The Los Alamos IPFT uses customized canister interfaces and external pumps to supply the system with measurement and clamping capabilities.

To evaluate a canister, the canister interface is placed over the filter on the canister lid. The IPFT establishes a suction seal to clamp the interface to the canister lid, while monitoring the suction pressure, and the test starts once the suction seal is established. Initially, the IPFT measures how much the internal system vacuum changes within the defined test period. If the filter is clogged, the measured air volume will be less than expected, and the system will attain the test pressure within a specified time. If the system does not reach the test pressure in that time frame, the filter is not clogged. A new "fill" time is defined based on the canister size, and a pressure decay test is then performed. For a 5-quart storage canister, the IPFT can measure an equivalent leak flow rate over a range from 0.03 to 3.0 cc/sec.

SPECIFICATIONS

Compatible containers: Hagan-style and SAVY-4000 Nuclear Material Storage Canisters

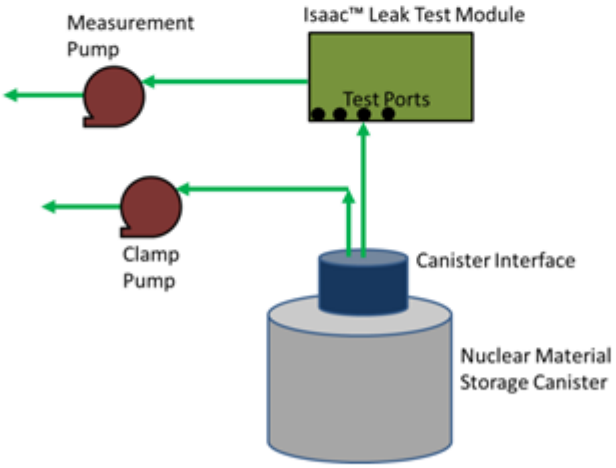
Pressure Decay Rate Ranges: 1.7E-03 to 1.7E-01 inWC/sec (Note: inWC = inches of water column)

Dimensions (L*W*H): Isacc® leak tester – 6.5*7*8.5 (inches), Canister interface – 2*2*2.75 (inches)

Weight: Total – 9.5 lbm, Isaac® leak tester – 6.9 lbm, Canister interface – 2.6 lbs

Power requirements: Isaac® 120 V at 60 Hz (2A) or 230 V at 50Hz (2A). External pumps are operated to manufacturer specifications.

Los Alamos IPFT Flow Schematic



Los Alamos National Laboratory
Radiation Protection Division
RP Services
Aerosol Engineering Facility
SM-30 Bikini Atoll MS G-761
Los Alamos, NM 87545

505-665-9661
memoore@lanl.gov
505-665-4827
adbrown@lanl.gov
505-6061585
kreeves@lanl.gov

B. POC component list

Item Description	Manufacturer/Supplier	Model Number	Cost	Quantity	URL
Flow meter	Honeywell	HAFBLF0750CAAX5	$96.08	1	http://www.digikey.com/product-detail/en/HAFBLF0750CAAX5/480-3868-ND/2839660
Pressure transducer (+/- 20" H2O)	Honeywell	SSCDRRN020NDAA5	$51.51	1	http://www.digikey.com/product-search/en?x=-851&y=-73&lang=en&site=us&KeyWords=SSCDRRN020NDAA5
Pressure transducer (30 PSI)	Honeywell	SSCDANN030PAAA5	$41.30	1	http://www.digikey.com/product-search/en?WT.z_header=search_go&lang=en&site=us&keywords=480-5392-5-ND&x=-762&y=-51
Pressure transducer (+/- 15 PSI)	Honeywell	SSCDANT015PGAA3	41.33	1	http://www.digikey.com/product-search/en?WT.z_header=search_go&lang=en&site=us&keywords=480-5392-5-ND&x=-762&y=-51
Solenoid valve	Clippard	MME-2QDS-D012	$20.49	4	http://www.clippard.com/part/MME-2QDS-W012
Vacuum pump	Clark Solutions	15988	$515.90	1	http://www.clarksol.com/html/prodspecs1500OPump.cfm
Sample pump	Clark Solutions	KPM-08A-3A	$18.00	1	http://www.clarksol.com/html/prodspecsKPMSPump.cfm
Suction head fitting	LANL Machine shop	In-house design	Unknown	2	
Microcontroller	Arduino	A000067	$50.57	1	http://www.digikey.com/product-detail/en/A000067/1050-1018-ND/2639006
Digital display/input shield	Adafruit	376	$59.00	1	http://www.adafruit.com/products/376
Relay board	SainSmart	8-Channel Relay Module	$18.04	1	http://www.amazon.com/SainSmart-8-CH-8-Channel-Relay-Module/dp/B0057OC5WK

C. Specification e-mail to Zaxis Inc for a custom Isaac™ unit

From: Moore, Murray E
Sent: Wednesday, May 07, 2014 4:17 PM
To: Harding, Doug
Cc: Brown, Austin D
Subject: May 7, 2014. Los Alamos (In-Place-Filter-Tester) specs (Moore)

Hi Doug,
Can you provide a quote for a system for the following specifications?

(1) The dead air space of the suction cup above the canister filter is 100 mL.
(2) We would like a test flow rate of 100 cc/min. Is this a variable quantity?
(3) We would like to leak test canisters in different sizes: 1, 3, 5, 8 and 12 quarts.
(4) We would like a clamp force on our suction cup of 80 inches of water. The clamp flow rate can be 100 cc/sec.
(5) How low is your resolution for leak rate? Our custom tester prototype has a baseline leak rate of 0.004 cc/sec.
(6) When the tested container has been measured as 'leaktight', the leak rate was about 0.3 cc/sec.
(7) During tests, we measure leak rates over a span from 0.3 cc/sec to about 10 cc/sec.

We will use an Alita pump (Alita model AL-80) from Tech Associates (Tech Assoc model # AP4V). (attached spec sheet). It has a dead-head vacuum of about 0.57 bars, and a free flow rate of 120 LPM.
Thank you for your help,
Murray

*

Los Alamos
NATIONAL LABORATORY
EST.1943

- Murray E. Moore, Ph.D., P.E.
- Desk phone: 505-665-9661
- R&D Engineer
- Los Alamos National Laboratory
- Radiation Protection Division, Group RP-SVS
- Aerosol Engineering Facility
http://www.lanl.gov/orgs/rp/wind_tunnel.shtml
http://int.lanl.gov/orgs/rp/rp2/wind_tunnel.shtml
** Shipping:*
SM-30 Bikini Atoll Rd. TA-3, Bldg-130, D.P. 01U,
Los Alamos NM 87545
- Mailing:
Group RP-SVS, Mail Stop G-761, Los Alamos NM 87545
~ Cell: 505-699-8264
~ Pager: 505-664-8140
~ Fax: 505-665-6071
~ Email: memoore@lanl.gov

MEDIUM CAPACITY OILLESS PUMP SERIES AP-3V, AP-4V, AP-6V, AP-8V

GREEN TURTLE PUMPS

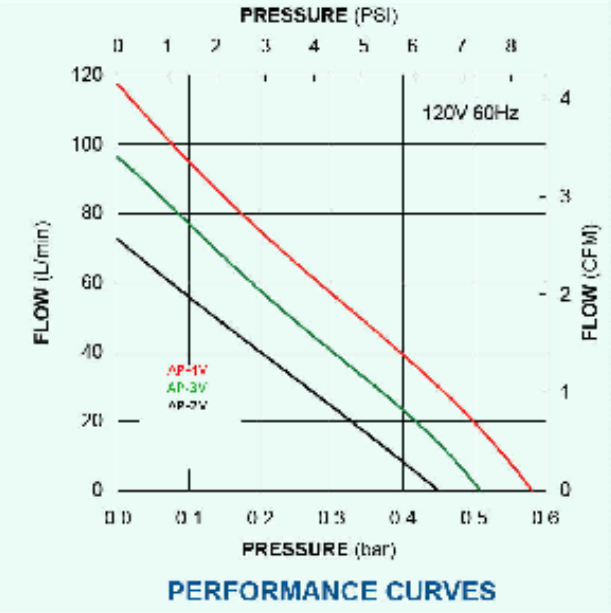
Very quiet- suitable for office use as well as industrial/medical locations.

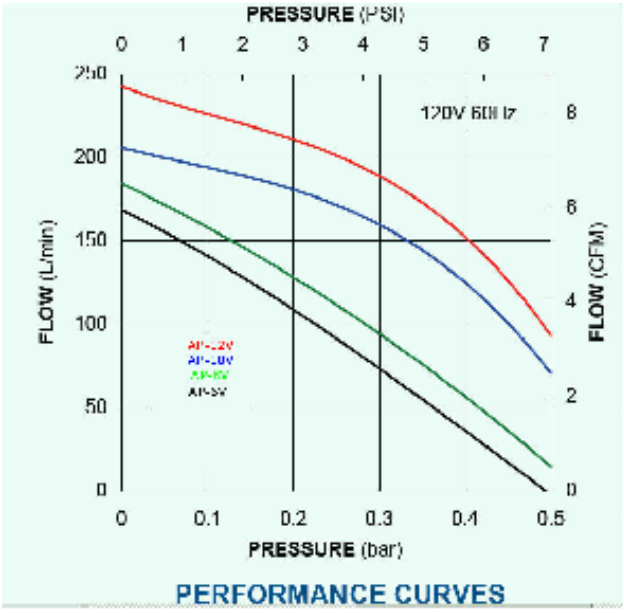
Model	AP2V		AP3V		AP4V	
Voltage, Freq	120V 60hZ	230v 50 Hz	120V 60hZ	230v 50 Hz	120V 60hZ	230v 50 Hz
Rated Pressure	0.15 bar (2.18psig)		0.15 bar (2.18psig)		0.15 bar (2.18psig)	
Rated Performance	48 lpm	40 lpm	68 lpm	60 lpm	85lpm	70 lpm
Rated Current Draw	0.8 A	0.5 A	1.0 A	0.6 A	1.7 A	0.8 A
Rated Input Power	48 W	40 W	60 W	54 W	85 W	80 W
Sound Level	36 dB		38 dB		36 dB	
Weight	5.6 kg (12.5 lbs)		6.5 kg (14.4 lbs)		6.5 kg (14.4 lbs)	
Dimension	205 mm L x 173 W x 202 mm H					

Model	AP6V		AP8V		AP10V		AP12V	
Voltage, Freq	120V 60h	230v 50 Hz	120V 60	230v 50 Hz	120V 60hZ	230v 50 Hz	120V 60hZ	230v 50 Hz
Rated Pressure	0.2 bar (2.9psig)							
Rated Performance	110 lpm	95 lpm	125 lpm	110 lpm	180lpm	170 lpm	210lpm	190 lpm
Rated Current Draw	2.1 A	1.0 A	2.2 A	1.1 A	2.3 A	1.2 A	3.2 A	1.3 A
Rated Input Power	126 W	115 W	132 W	126 W	138 W	138 W	250 W	200 W
Sound Level	41 dB		42 dB		44 dB		45 dB	
Weight	8.3 kg (18.2 lbs)		8.3 kg (18.2 lbs)		10.1 kg (22.3 lbs)		10.1 kg (22.3 lbs)	
Dimension	248 mm L x 205 W x 245 mm H							
Information presented is based on test results from nominal units								
Specifications are subject to change without notice								

TA TECHNICAL ASSOCIATES

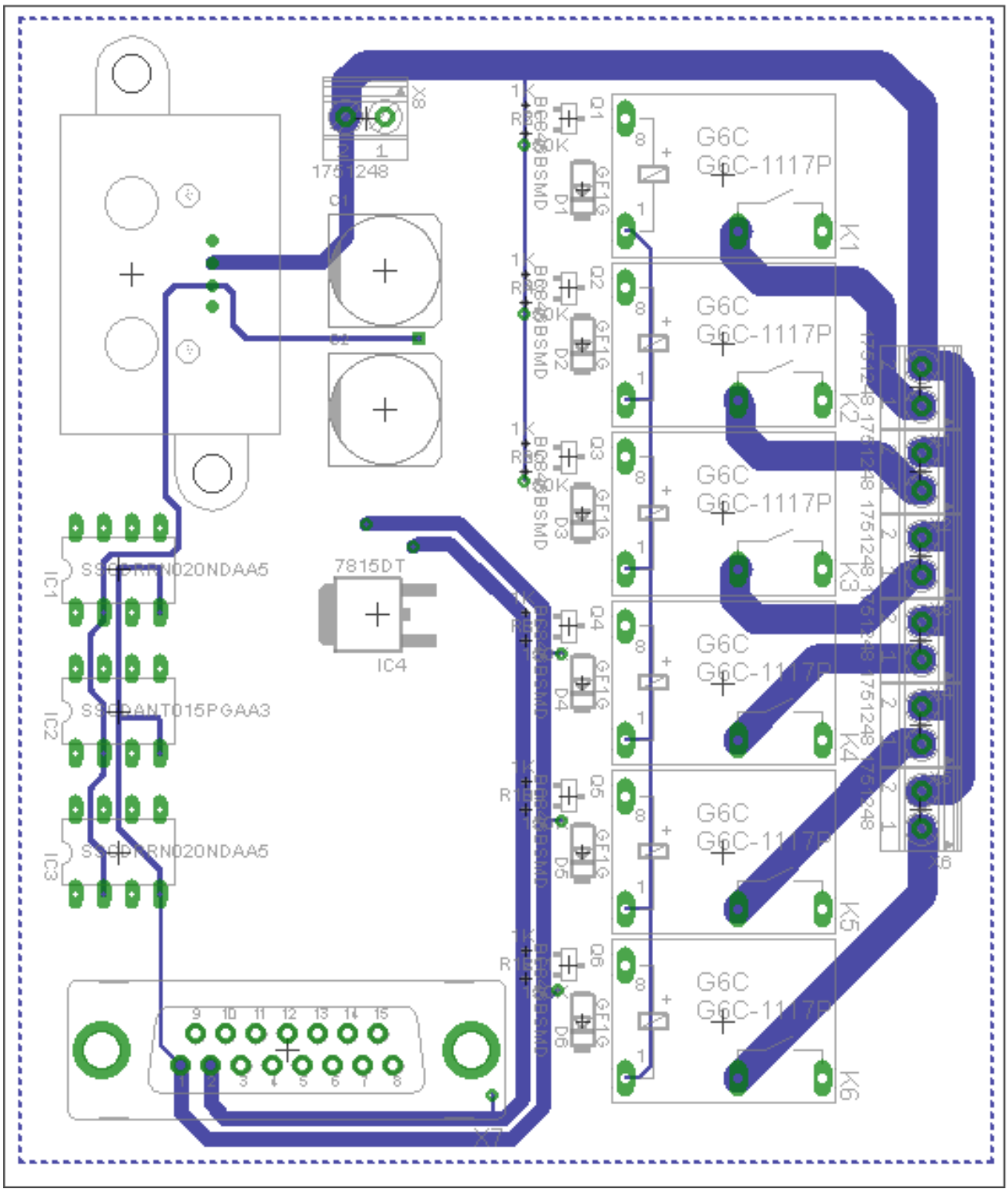
7061 ETON AVENUE, CANOGA PARK, CA. 91303
TELEPHONE (8 [illegible] - FAX (818) 882-2100
e-mail: tagold@aol.com - www.tech-associates.com

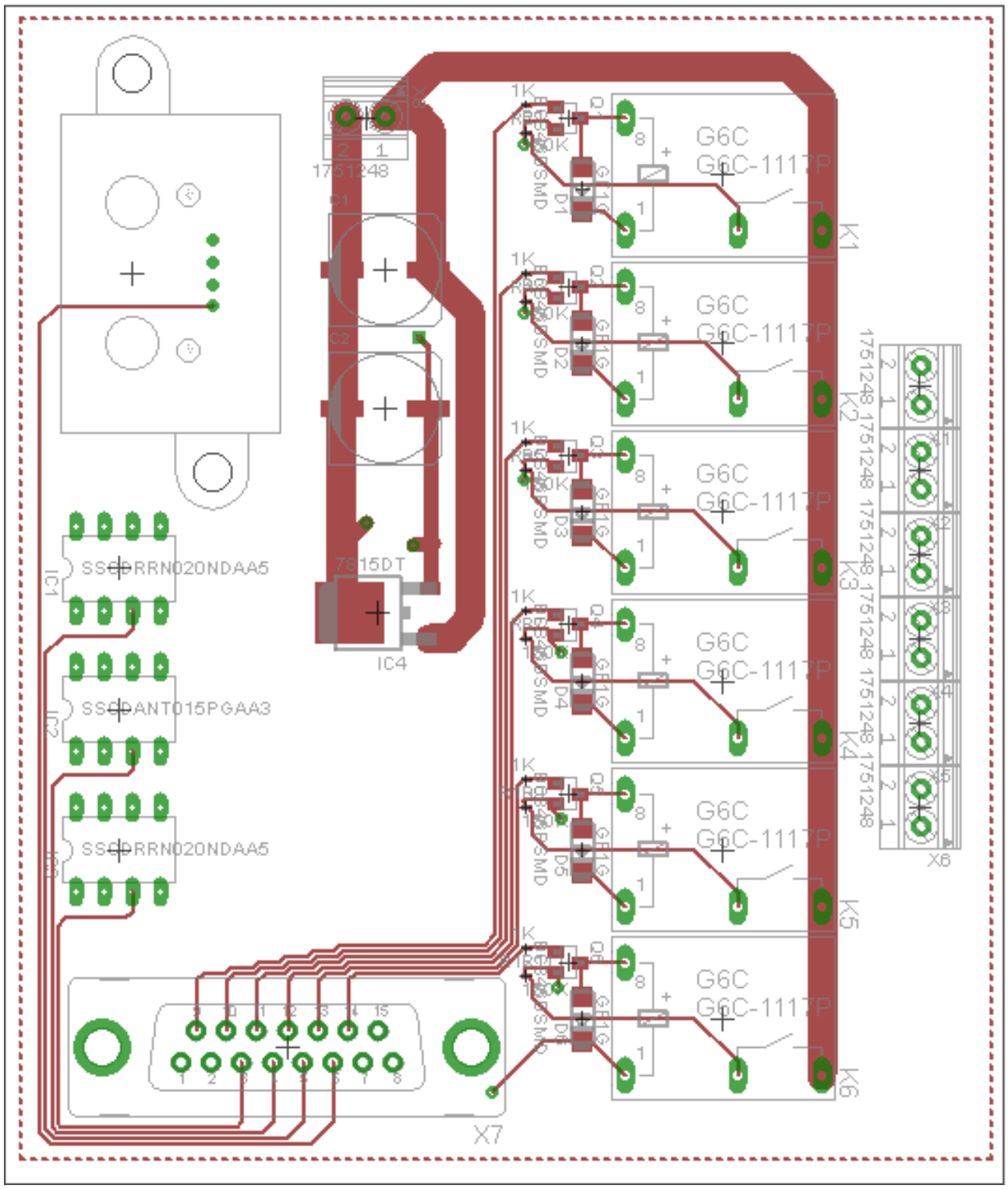


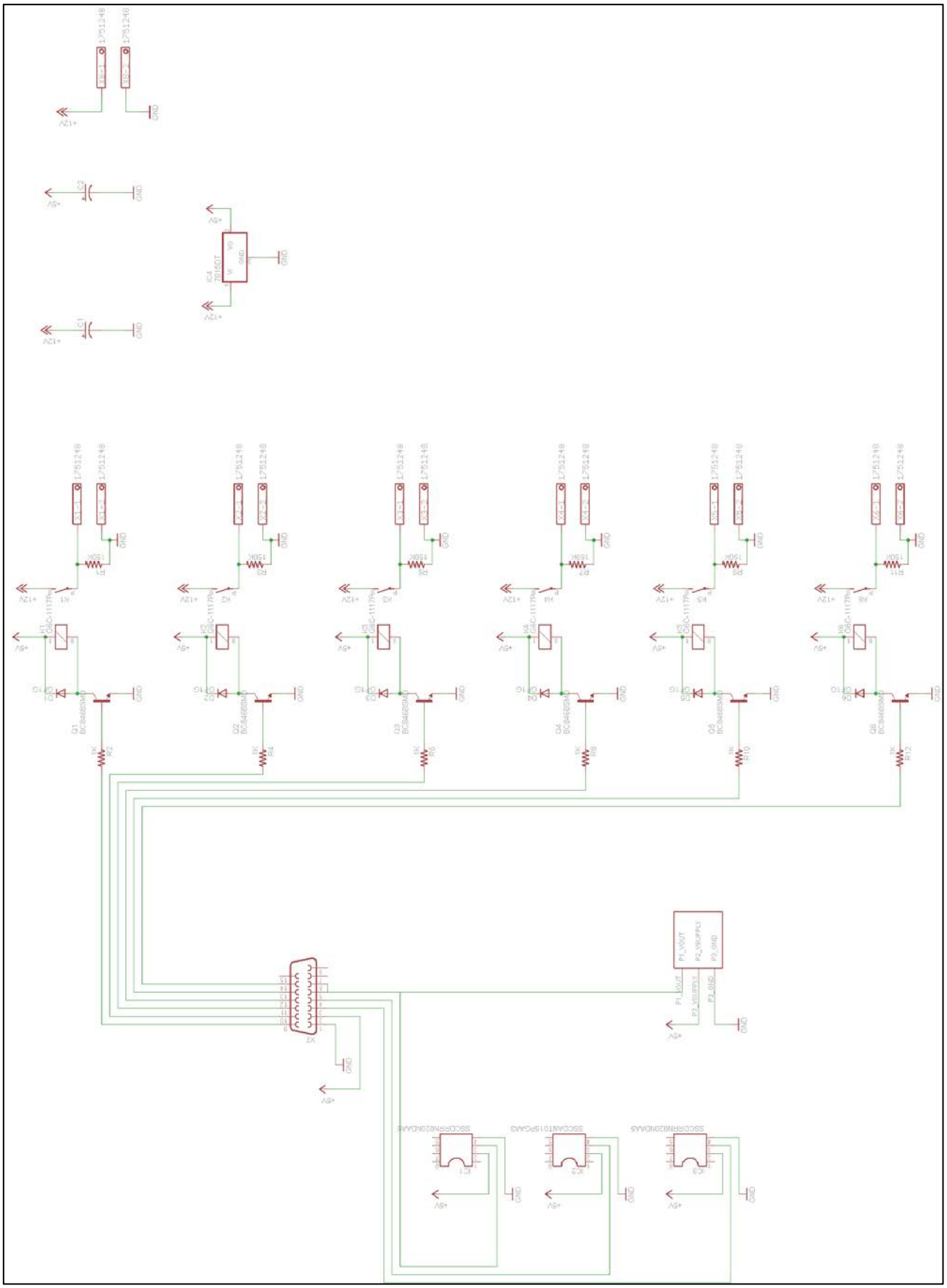


TA TECHNICAL ASSOCIATES
7000 ETON AVENUE, CANOGA PARK, CA 91303
TELEPHONE (818) 882-0100 • FAX (818) 882-0109

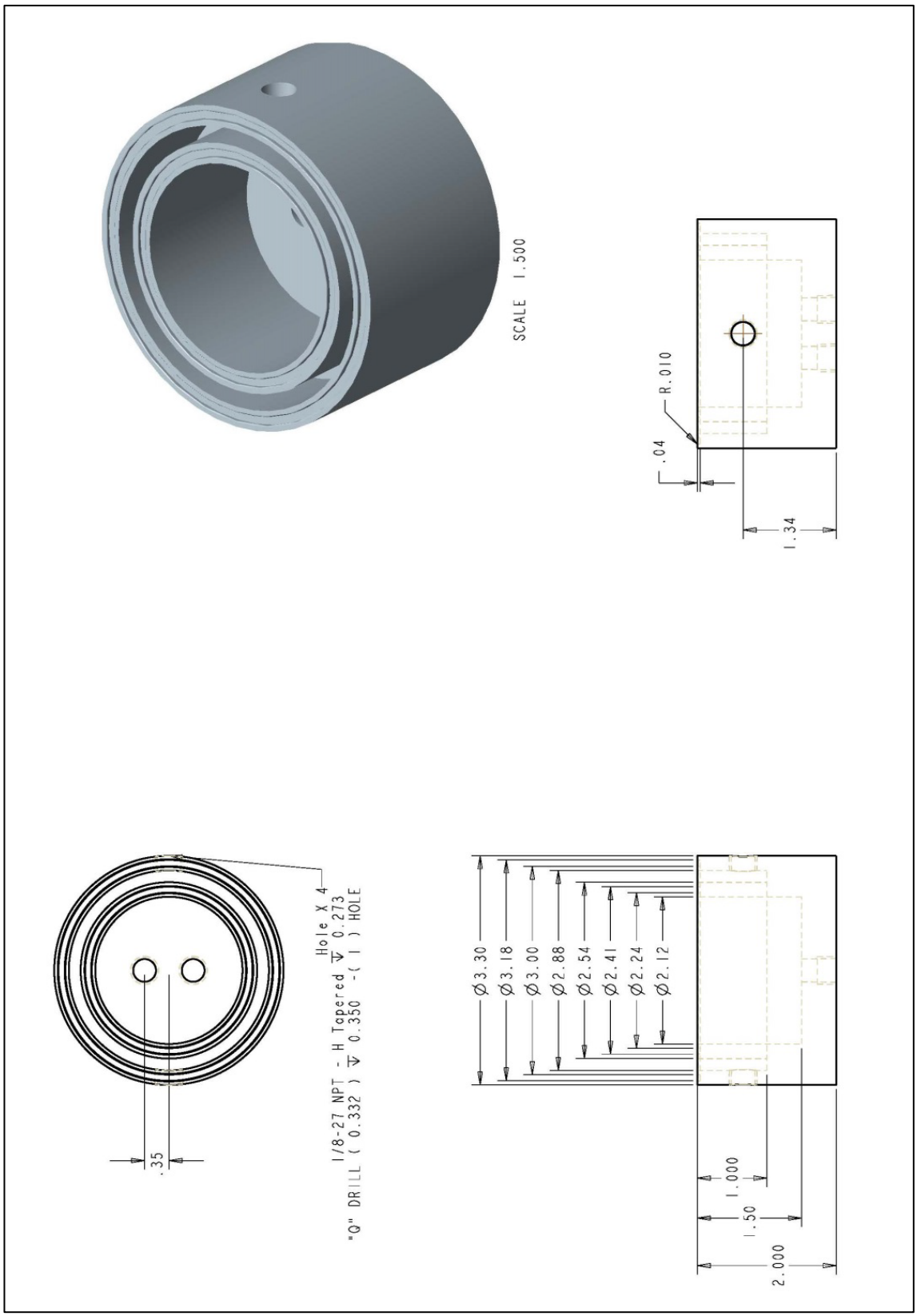
D. LANL PCB – Printed Circuit Board

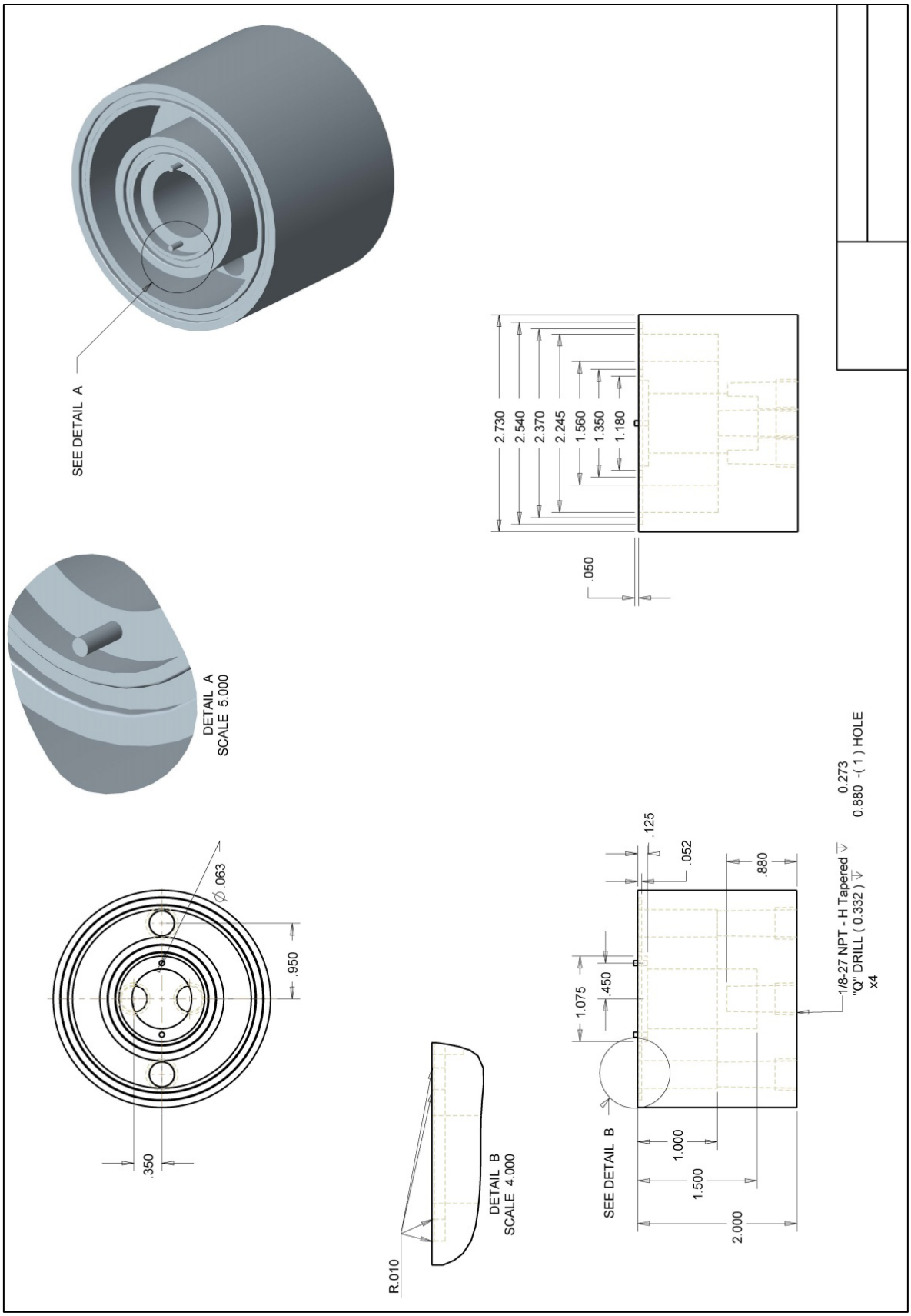



G6C
G6C-1117P
K1
K2
K3
K4
K5
K6
X6
X7
IC1
IC2
IC4
7815DT
1751248

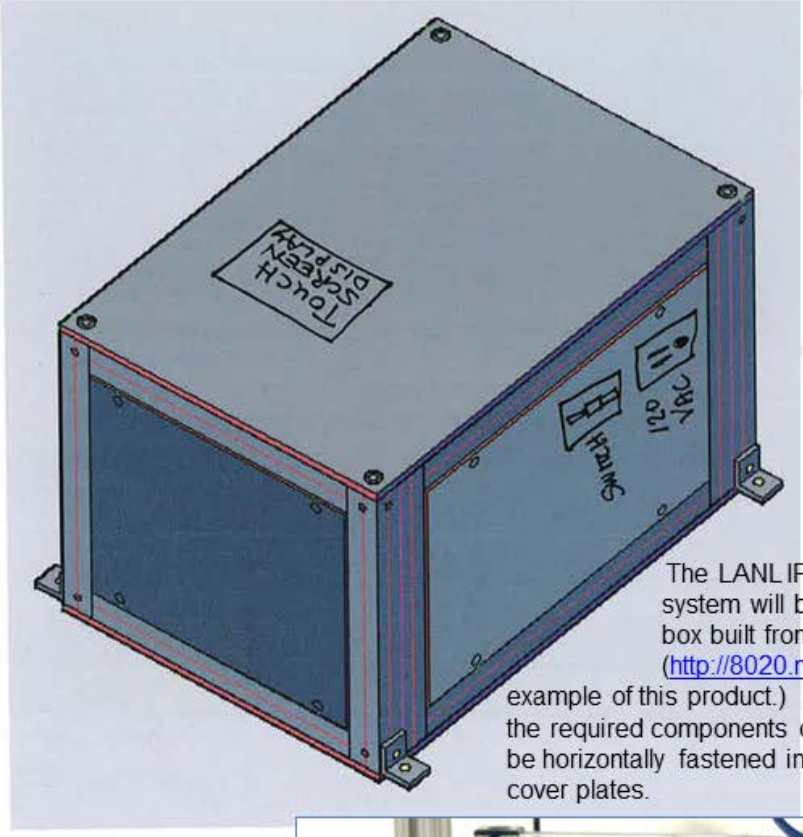


E. Canister Interface




SEE DETAIL A
2.730
2.540
2.370
2.245
1.560
1.350
1.180
.050
DETAIL A
SCALE 5.000
Ø .063
.950
.350
.125
.052
.880
1.075
.450
1/8-27 NPT - H Tapered
"Q" DRILL (0.332)
x4
0.273
0.880 -(1) HOLE
SEE DETAIL B
1.000
1.500
2.000
R.010
DETAIL B
SCALE 4.000

F. Build Request to vendor of IPFT device



Los Alamos NATIONAL LABORATORY EST.1943

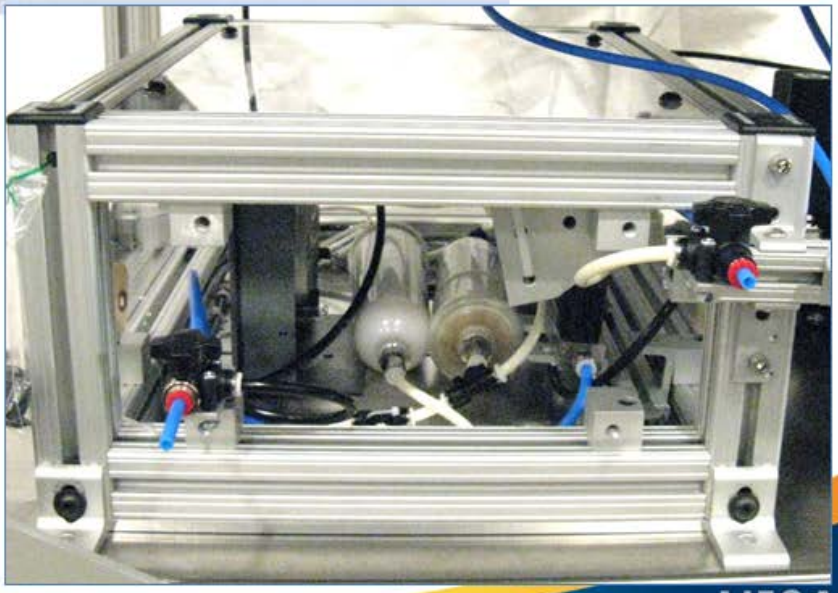
Murray E. Moore
505-665-9661
Austin D. Brown
505-665-4827

Los Alamos National Labs

The LANL IPFT (In-Place-Filter-Tester) system will be built on the platform of a box built from 8020™ components. (http://8020.net/). (See photo below for an example of this product.) However, Betatron will mount the required components on an internal plate that will be horizontally fastened in between the top and bottom cover plates.

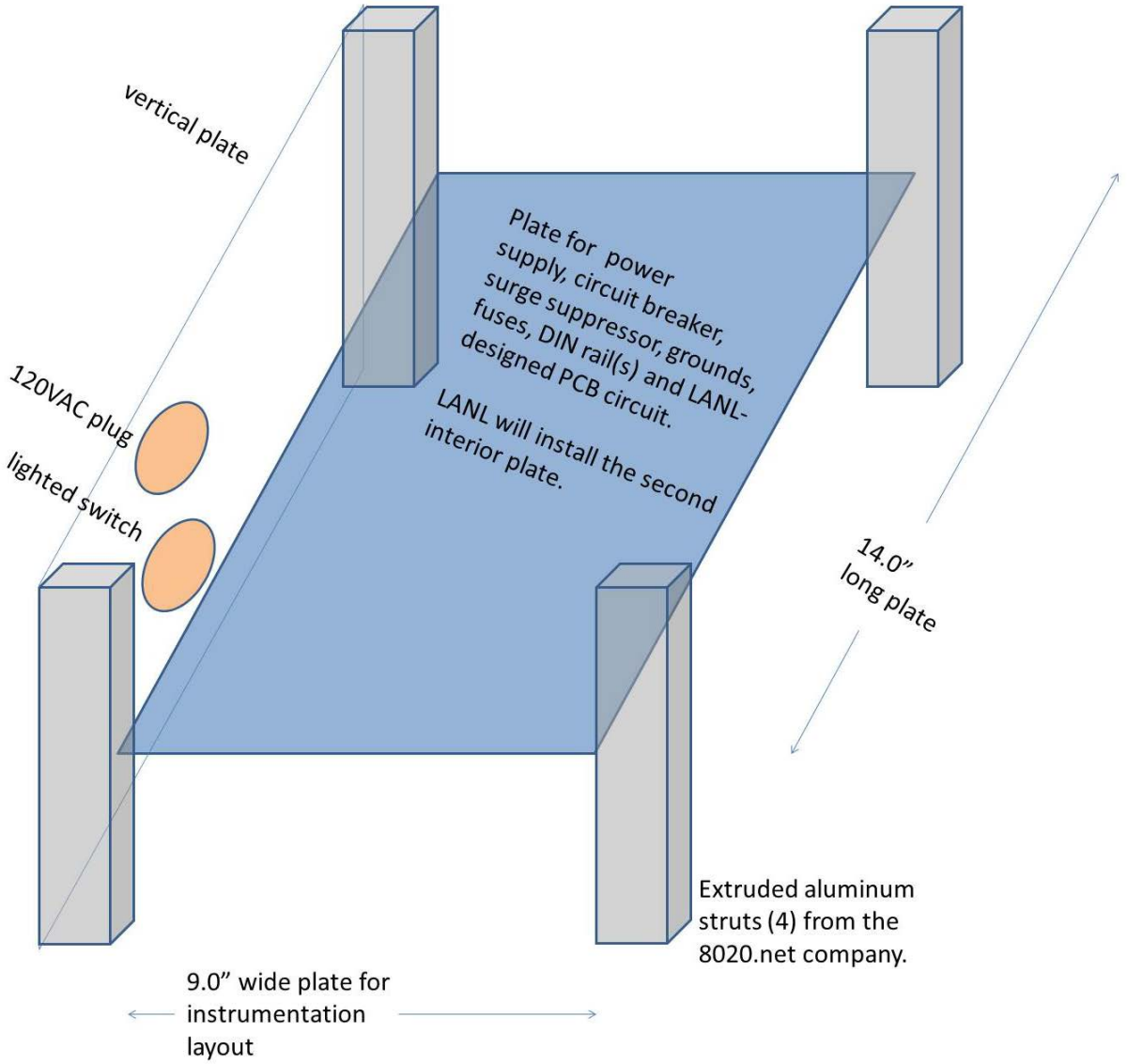
Randy Brown
HeiTek Automation
21602 N 2nd Ave Suite 4
Phoenix, AZ 85027
602-269-7931 office
rbrown@heitek.com
Automate or Evaporate

This is the contact info for the 8020.net supplier.



Vendor would supply two(2) of these units:

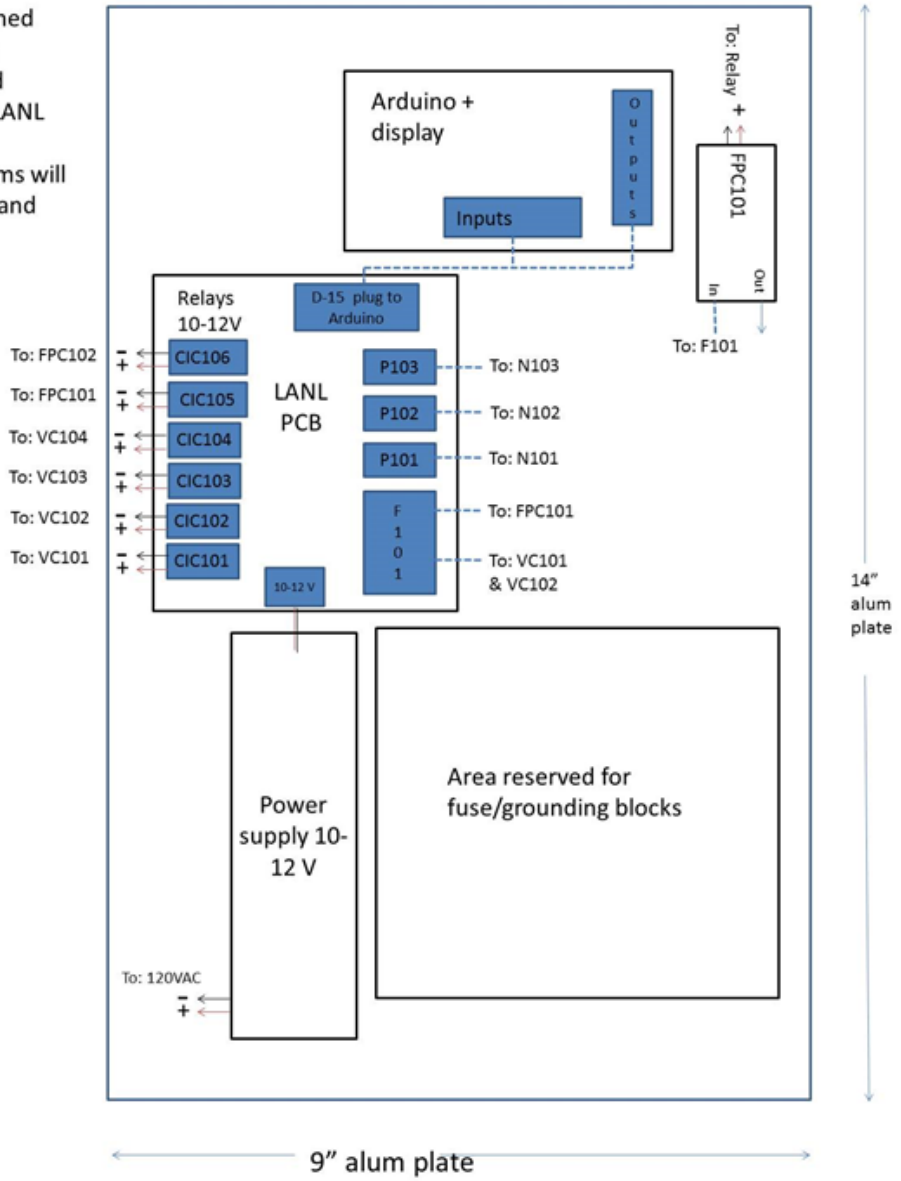
Instrument plate with the indicated items installed. The assembly does NOT have to be UL listed or QA/QC approved. This sketch indicates only two of the eight total aluminum plates from the 8020.net package. The four pillars are shown for sake of mechanically connecting the two pates together. There are six exterior plates and two internal plates. See attachments.



Electronics side of the 9" X 14" Plate PHYSICAL LAYOUT

All Dashed lined items will be provided and installed by LANL

Solid line items will be provided and installed by Betatron



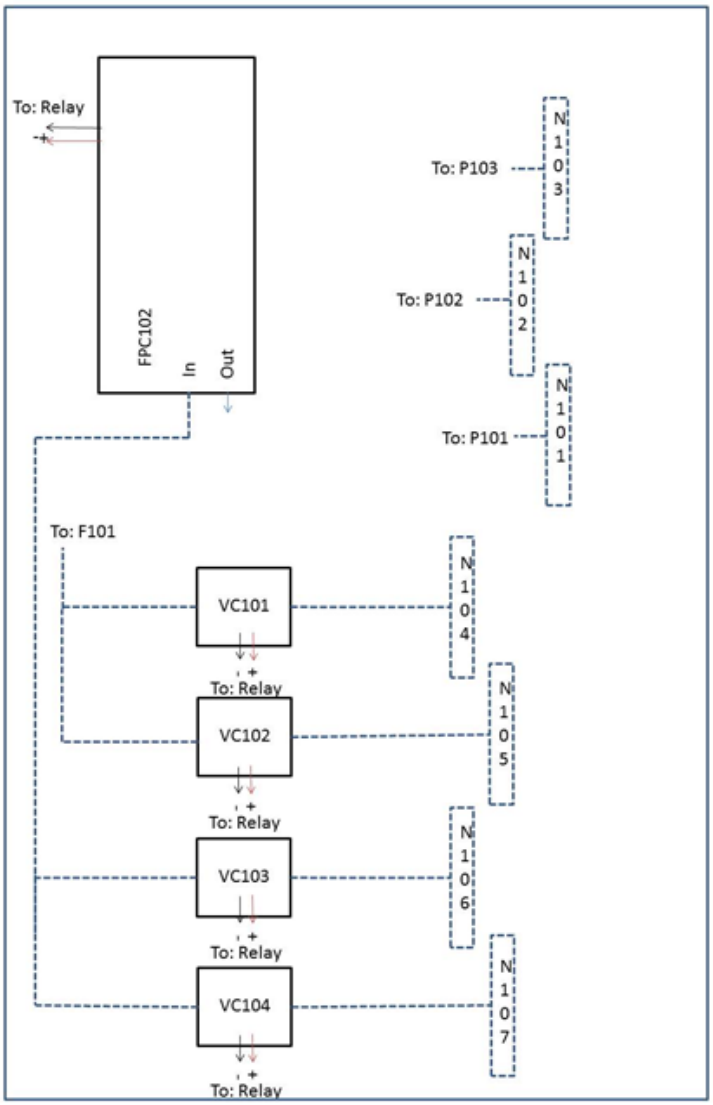
Aluminum plate (8 gauge) . Both sides will be used for instrument mounting. Supplied with box components from Heitek.

Los Alamos National Laboratory – ME Moore , 505-665-9661 and AD Brown 505-665-4827 IPFT – Betatron prototype

Pump side of the 9" X 14" Plate PHYSICAL LAYOUT

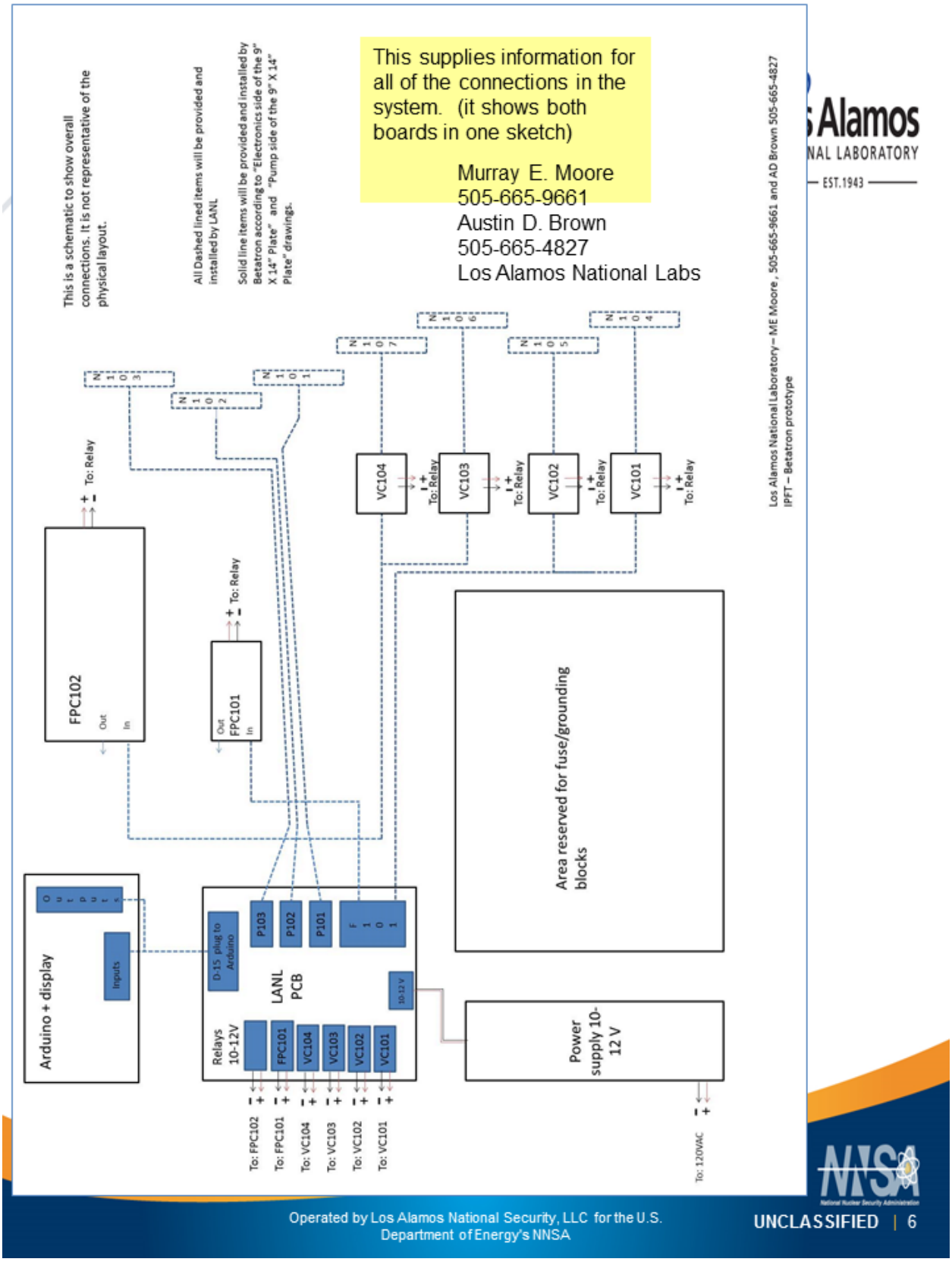
All Dashed lined items will be provided and installed by LANL

Solid line items will be provided and installed by Betatron



Aluminum plate (8 gauge) . Both sides will be used for instrument mounting. Supplied with box components from Heitek.

Los Alamos National Laboratory – ME Moore , 505-665-9661 and AD Brown 505-665-4827 IPFT – Betatron prototype


This supplies information for all of the connections in the system. (it shows both boards in one sketch)
Murray E. Moore
505-665-9661
Austin D. Brown
505-665-4827
Los Alamos National Labs
This is a schematic to show overall connections. It is not representative of the physical layout.
All Dashed lined items will be provided and installed by LANL
Solid line items will be provided and installed by Betatron according to "Electronics side of the 9" X 14" Plate" and "Pump side of the 9" X 14" Plate" drawings.
Los Alamos National Laboratory – ME Moore , 505-665-9661 and AD Brown 505-665-4827
IPFT – Betatron prototype
FPC102
FPC101
Out
In
To: Relay
VC104
VC103
VC102
VC101
Area reserved for fuse/grounding blocks
Arduino + display
Outputs
Inputs
D-15 plug to Arduino
LANL PCB
P103
P102
P101
F101
10-12 V
Relays 10-12V
FPC101
VC104
VC103
VC102
VC101
Power supply 10-12 V
To: FPC102
To: FPC101
To: VC104
To: VC103
To: VC102
To: VC101
To: 120VAC

Notes on Betatron layout drawing

- Betatron should quote for a quantity of two (2) systems.
- Betatron should design this to be UL listed.
- Betatron should design fusing and power routing systems.
- Betatron should notice the aluminum plates for the main instrument installation and power cord plug are part of a package quote from the Heitek company (http://heitek.com/) Tel 800-926-2099 contact: Randy Brown. Betatron should receive the Heitek quote from LANL or from Heitek. All extra parts in the quote package are LANL property.
- Betatron should purchase the box from the Heitek company directly based on the quote.
- Betatron should notice that the drawing is to scale, and all devices should be located in close proportion to their positions in the sketch.
- Betatron should specify all bracket holes and screw mounts.
- Betatron should specify and place fan/vent.
- Betatron should provide phenolic label "LANL In-Place-Filter-Tester for SNM canisters" (0.25" high letters).
- Betatron should install the FPC102 (larger pump) with a supplied bracket.
- Betatron should purchase the power supply (designation is Phoenix #2866488).
- Betatron should install the external 120VAC power cord (it's removable like a desktop PC).
- Betatron should ensure the "bottom" and "south side" are kept clear so that the box can lay flat or stand on edge.
- Betatron should assemble the system box to include at least the main internal instrument mounting plate and the side plate for the 120VAC connection. If they need to attach the internal support pillars for ease of fabrication, that should be done.
- Betatron should design, specify and cut the required hole in the top cover plate for the touchscreen (above the Arduino™ board). It should be a flush-mount support assembly. The platform should be designed and installed by Betatron.

- LANL should connect pneumatic lines for pumps valves and filters.
- LANL should specify the enclosure box and mounting plate (this is the Heitek quote, it's been done already).
- LANL should design and drill bulkhead holes.
- LANL should supply the D-15 cable which connects the LANL PCB to the Arduino board. It should be created and installed by LANL.

- Los Alamos National Laboratory ME Moore , 505-665-9661 and AD Brown 505-665-4827

UNCLASSIFIED

Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA

UNCLASSIFIED | 7

P&ID component list

CI101 Computer indication link to pressure transducer PI-101
CI102 Computer indication link to pressure transducer PI-102
CI103 Computer indication link to pressure transducer PI-103
CI104 Computer indication link to flow meter FI-101

CIC101 Computer control link to VC 101
CIC102 Computer control link to VC 102
CIC103 Computer control link to VC 103
CIC104 Computer control link to VC 104
CIC105 Computer control link to FPC 101
CIC106 Computer control link to FPC 102

FI101 Mass flow meter

FPC101 Measurement pump
FPC102 Suction force clamping pump

N101 - N107 Filter, HEPA

P101 Adapter to SNM canister with two measurement ports
P102 Adapter to SNM canister with two suction force clamping ports
P103 SNM canister with filter in lid

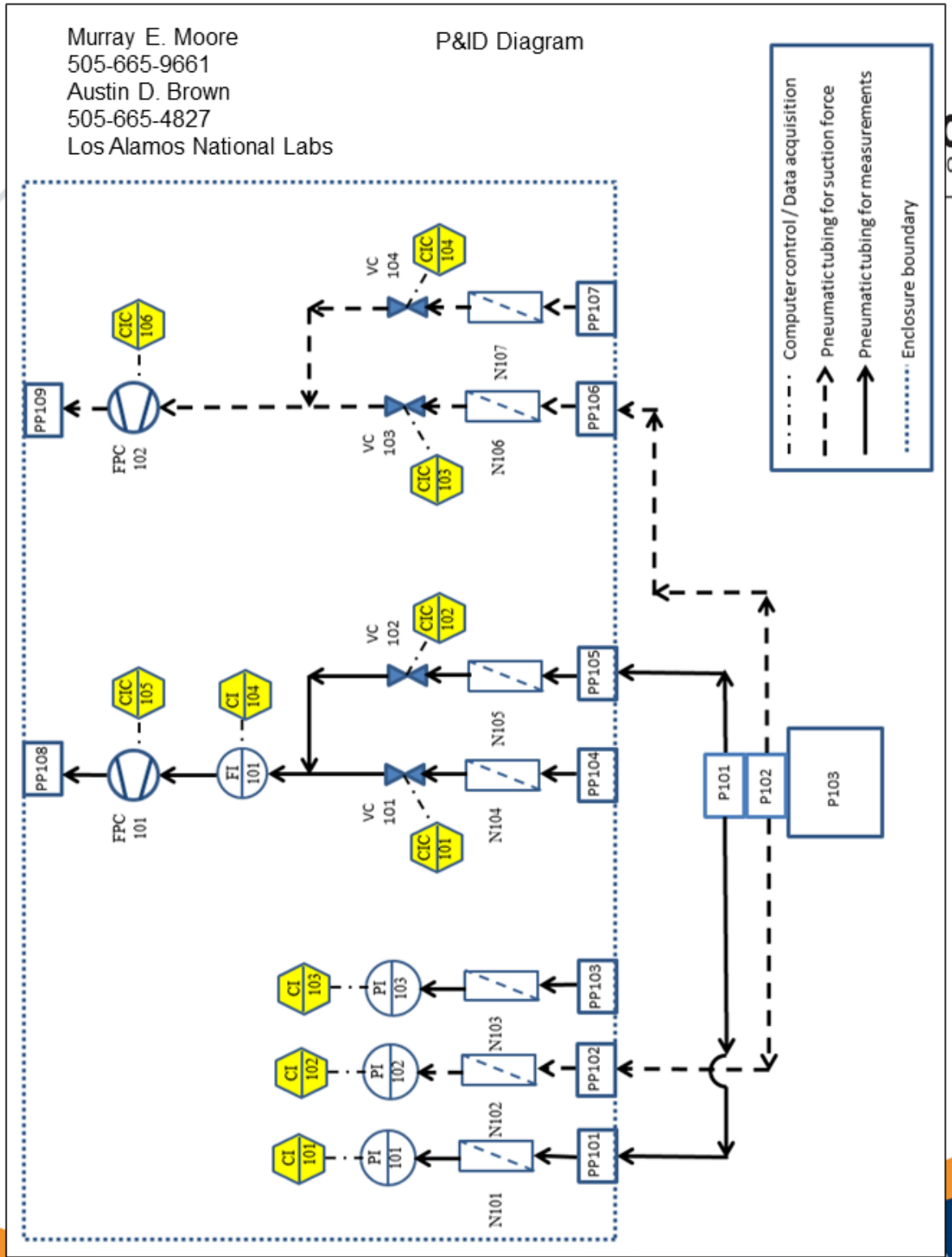
PI101 Differential pressure gauge
PI102 Differential pressure gauge
PI103 Absolute pressure gauge

PP101 - PP109 Bulkhead for vacuum test / pressure relief connections

VC101 - VC104 Two-way solenoid valve

POWER SUPPLY: Designation is Phoenix #2866488.

Murray E. Moore
505-665-9661
Austin D. Brown
505-665-4827
Los Alamos National Labs


Murray E. Moore
505-665-9661
Austin D. Brown
505-665-4827
Los Alamos National Labs
P&ID Diagram
Computer control / Data acquisition
Pneumatic tubing for suction force
Pneumatic tubing for measurements
Enclosure boundary
CIC 104
VC 104
PP107
CIC 106
N107
PP109
PP106
FPC 102
VC 103
CIC 103
N106
CIC 102
VC 102
PP105
CIC 105
CI 104
N105
PP108
FI 101
PP104
P101
P102
P103
FPC 101
VC 101
CIC 101
N104
CI 103
PI 103
PP103
N103
CI 102
PI 102
PP102
N102
CI 101
PI 101
PP101
N101

G. Pressure drop reconciliation

From: Moore, Murray E

Sent: Thursday, August 15, 2013 1:39 PM

To: Veirs, Douglas K; Smith, Paul Herrick; Kirk Reeves

Subject: Pressure drop measurement - reconciled. (LANL vs. NucFilt filter pressure drop)

Hello all,

I am happy to announce that we have reconciled the pressure drop measurement discrepancy that happened during a set of paired comparison tests.

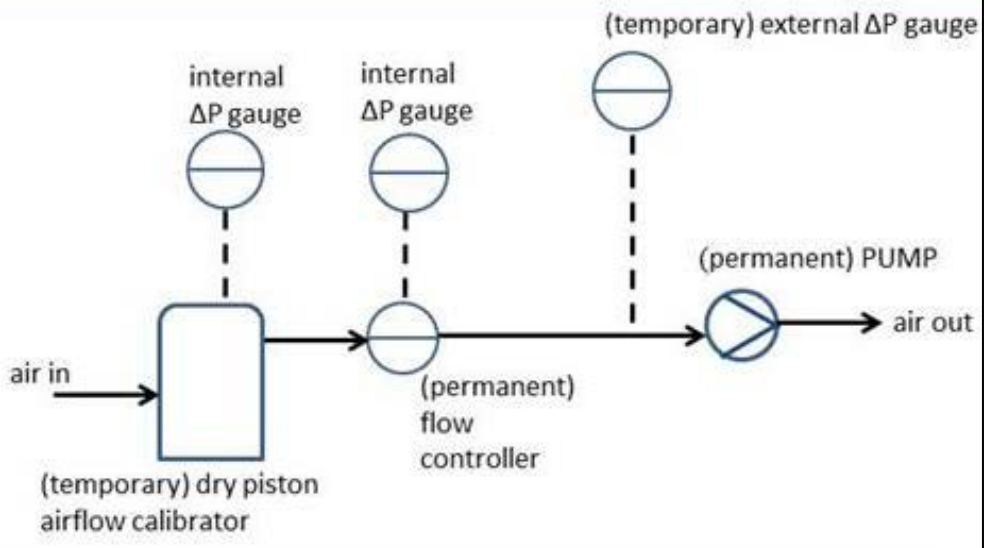
(1) **Previously, the mistake made** on 5-30-2013 (p.81 logbook LANL-NucFilt Moore #4) involved using the volumetric flowrate value of 200 accm as the setpoint for the flow controller. This is incorrect, since the flow controller measures the internal volumetric flow (downstream of its transducer), this is NOT the same as the volumetric flow that enters into the filter test system and passes through the filter.

In summary, from this day forward, to establish a flow rate of 200 accm through a test filter:

(1) Set the **mass** flowrate through the flow controller FIC102 to be 152 sccm (mass flow).
(2) When the flow controller has an established flow of 152 sccm, a typical absolute pressure in the flow controller will be about 8.49 psia (internal pressure, downstream of the flow controller), and the flow controller will indicate about 263 accm (volumetric flow).
(3) To verify the system operation, a dry piston primary calibrator is connected to the inlet port of the flow controller. (On August 15, 2013, the corresponding mass flowrate through the dry piston calibrator was 151 sccm (mass flow)). The dry piston calibrator display indicates that the 152 sccm in mass flow is equivalent to 200 accm in volumetric flow.
(4) **The required volumetric flow (200 accm) has to be measured at the inlet to the system, and expressed in terms of mass flow (152 sccm**).

152 sccm / 200 accm = 0.76 (the local Los Alamos pressure ratio)

(5) The mass flow of 152 sccm must be used as the setpoint for the flow controller.
(6) In reference to the included sketch, the mass flowrate through the system remains constant, but the volumetric flowrate does not. The volumetric flowrate is locally defined, based on the local air pressure.



Schematic of calibration verification flow layout. "Permanent" items are part of the LANL SNMC filter test system. "Temporary" items are not part of the operating layout. The mass flow values match, but the correct value of 200 accm is only indicated when a temporary calibration flowmeter is utilized.

Indicated measured values	Omega™ flow controller	Bios™ Dry Piston Calibrator (temporary)	Pressure Gauge (temporary)
Volumetric flow (accm)	263	200	N/A
Mass Flow (sccm)	**152**	**151**	N/A
Pressure (psia)	8.49	11.2	8.52

We were able to measure the pressure drop across one of the reference Hagan-type filters at the measured mass flowrate of 152 sccm (corresponding to the correct 200 accm through the filter-that-was-tested).

The measured pressure drop matched the value that was previously measured by NFT Inc (about 0.8 inches of water column, in W.C.).

To be done: a full measurement of all the reference filters (about 20 total) with the correct air flowrate value and the correct aerosol size distribution.

*

- Murray E. Moore, Ph.D., P.E.

- Desk phone: 505-665-9661

H. NFT reference filter standards

NFT
741 Corporate Circle, Suite R
Golden, Colorado 80401
Phone: 303.384.9785
Fax: 303.384.9579
NFTINC.com

NFT
DOCUMENTATION PACKAGE
COVER LETTER

CUSTOMER: Los Alamos National Security
PO NUMBER: 243568
MODEL/DESCRIPTION: SAVY-4000 Filter Standards
SPECIFICATION: Contract #243568 – Exhibit D
DRAWING: 20100220 Rev 1
DATE: October 16, 2013

PS6
9/02

Nuclear Filter Technology
Filter Efficiency Test Data
(Minimum Capacity and HEPA Particulant)

WO.#: 4222-001 C		Filter Media:	CERAMIC
Date of Manufacture:	10/13	Filter Housing Type:	STAINLESS STEEL
Sealant Type:	N/A	Lid Type:	STAINLESS STEEL
Customer I.D.	243568-2L-001	THRU	243568-2L-003
QA Engineer 10-16-13		Test Engineer	10-16-13

SERIAL NUMBER	Flow 200 ±10 ml/min	% Penetration Max. = 0.025	Resistance Max. = 1
243568-2L- 001	200	0.0001	0.64
243568-2L- 002	200	0.0001	0.66
243568-2L- 003	200	0.0001	0.66

All tests are conducted according to Nuclear Filter Technology procedure PS 6.

NFT
741 Corporate Circle, Suite R
Golden, Colorado 80401
Phone: 303.384.9785
Fax: 303.384.9579
NFTINC.com

NFT
HYDROGEN DIFFUSION CALCULATION SUMMARY FOR:
SAVY-4000 Filter Standards
SPECIFICATION: Contract #243568 – Exhibit D

CUSTOMER: Los Alamos National Security

P.O.#: 243568

MEASURED DIFFUSION COEFFICIENT AT 25° C Must be ≥ 2.4 E-5 MOL/SEC/MOL FRAC.	
TEST NUMBER	**MEASURED**
243568-2L-002 WO 4222-001 C	7.62E-05

HYDROGEN DIFFUSION TESTS ARE PERFORMED USING QUALIFIED TEST PROCEDURE PS 26.

10.16.13
QA Manager Date

Attachment 1 – Test Data Form

Test Pressure (psig/wc):	12"-14" WC	Item ID:	Filter Standards
Test Time (seconds):	60 Seconds	Batch/WO No.:	4222-001C
Pressure Gauge ID:	PG 148	Pressure Gauge Cal. Due:	5/6/2014
Pressure Gauge Range:	0"-15" WC		
Setup Verification (sign):	Paul Farnsworth	Date:	10/10/2013

Entry No:	Test Article ID	Accept / Reject (A/R)	Entry No:	Test Article ID	Accept / Reject (A/R)
1	243568-2L-001	A	26		
2	243568-2L-002	A	27		
3	243568-2L-003	A	28		
4			29		
5			30		
6			31		
7			32		
8			33		
9			34		
10			35		
11			36		
12			37		
13			38		
14			39		
15			40		
16			41		
17			42		
18			43		
19			44		
20			45		
21			46		
22			47		
23			48		
24			49		
25			50		

Attachment 1 - Test Data Form Continued

COMMENTS Page 2 of 2

Test Performed By: [signature]	Certification Type/Level (If Applicable) 2	Test Date: 10-10-13
Quality: Paul Farnsworth	Certification Type/Level (If Applicable)	Date: 10/10/2013

NFT
741 Corporate Circle, Suite R
Golden, Colorado 80401
Phone: 303.384.9785
Fax: 303.384.9579
NFTINC.com

PACKING LIST

From

NFT
741 CORPORATE CIRCLE, SUITE R
GOLDEN, CO 80401
USA

DATE:	October 16, 2013
SOLD TO:	LOS ALAMOS NATIONAL SECURITY
SHIPPING ADDRESS:	LANL FOR US DOE / NNSA SM-30 WAREHOUSE BIKINI ATOLL ROAD LOS ALAMOS, NM 87545
SHIP VIA:	**BEST WAY, FREIGHT PRE-PAID AND ADDED TO INVOICE**
CUSTOMER P.O.:	**243568**
BILLING ADDRESS:	LOS ALAMOS NATIONAL SECURITY invoices@lanl.gov
QUANTITY:	3
DESCRIPTION:	SAVY-4000 Filter Standards
SERIAL #:	243568-2L-001 thru 243568-2L-003
DATE OF MANUFACTURE:	10/13
NUMBER OF BOXES:	_____

NFT
741 Corporate Circle, Suite R
Golden, Colorado 80401
Phone: 303.384.9785
Fax: 303.384.9579
NFTINC.com

NFT
DOCUMENTATION PACKAGE
COVER LETTER

CUSTOMER: Los Alamos National Security
PO NUMBER: 243568
MODEL/DESCRIPTION: SAVY-4000 Filter Standards
SPECIFICATION: Contract #243568 – Exhibit D
DRAWING: 20100230 Rev 1
DATE: October 16, 2013

PS6
9/02

Nuclear Filter Technology
Filter Efficiency Test Data
(Minimum Capacity and HEPA Particulant)

WO.#:	4222-001 B	Filter Media:	CERAMIC
Date of Manufacture:	10/13	Filter Housing Type:	STAINLESS STEEL
Sealant Type:	N/A	Lid Type:	STAINLESS STEEL
Customer I.D.	243568-3L-001	THRU	243568-3L-012
QA Engineer	10-16-13	Test Engineer	10.16.13

SERIAL NUMBER	Flow 200 ±10 ml/min	% Penetration Max. = 0.025	Resistance Max. = 1
243568-3L- 001	200	0.0001	0.77
243568-3L- 002	200	0.0001	0.76
243568-3L- 003	200	0.0001	0.78
243568-3L- 004	200	0.0001	0.82
243568-3L- 005	200	0.0009	0.73
243568-3L- 006	200	0.0001	0.72
243568-3L- 007	200	0.0001	0.82
243568-3L- 008	200	0.0001	0.74
243568-3L- 009	200	0.0001	0.91
243568-3L- 010	200	0.0001	0.68
243568-3L- 011	200	0.0001	0.82
243568-3L- 012	200	0.0001	0.87

All tests are conducted according to Nuclear Filter Technology procedure PS 6.

NFT
741 Corporate Circle, Suite R
Golden, Colorado 80401
Phone: 303.384.9785
Fax: 303.384.9579
NFTINC.com

NFT
HYDROGEN DIFFUSION CALCULATION SUMMARY FOR:
SAVY-4000 Filter Standards
SPECIFICATION: Contract #243568 – Exhibit D

CUSTOMER: Los Alamos National Security

P.O.#: 243568

MEASURED DIFFUSION COEFFICIENT AT 25° C Must be ≥ 2.4 E-5 MOL/SEC/MOL FRAC.	
TEST NUMBER	**MEASURED**
243568-3L-006 WO 4222-001 B	7.04E-05

HYDROGEN DIFFUSION TESTS ARE PERFORMED USING QUALIFIED TEST PROCEDURE PS 26.

10.16.13
QA Manager Date

Attachment 1 – Test Data Form

Test Pressure (psig/wc):	12"-14" WC	Item ID:	Filter Standards
Test Time (seconds):	60 Seconds	Batch/WO No.:	4222-001B
Pressure Gauge ID:	PG 148	Pressure Gauge Cal. Due:	5/6/2014
Pressure Gauge Range:	0"-15" WC		
Setup Verification (sign):	Paul Farnsworth	Date:	10/10/2013

Entry No:	Test Article ID	Accept / Reject (A/R)	Entry No:	Test Article ID	Accept / Reject (A/R)
1	243568-3L-001	A	26		
2	243568-3L-002	A	27		
3	243568-3L-003	A	28		
4	243568-3L-004	A	29		
5	243568-3L-005	A	30		
6	243568-3L-006	A	31		
7	243568-3L-007	A	32		
8	243568-3L-008	A	33		
9	243568-3L-009	A	34		
10	243568-3L-010	A	35		
11	243568-3L-011	A	36		
12	243568-3L-012	A	37		
13			38		
14			39		
15			40		
16			41		
17			42		
18			43		
19			44		
20			45		
21			46		
22			47		
23			48		
24			49		
25			50		

Attachment 1 - Test Data Form Continued

COMMENTS Page 2 of 2

Test Performed By: [signature]	Certification Type/Level (If Applicable) 2	Test Date: 10-10-13
Quality: Paul Farnsworth	Certification Type/Level (If Applicable)	Date: 10/10/2013

NFT
741 Corporate Circle, Suite R
Golden, Colorado 80401
Phone: 303.384.9785
Fax: 303.384.9579
NFTINC.com

PACKING LIST

From

NFT
741 CORPORATE CIRCLE, SUITE R
GOLDEN, CO 80401
USA

DATE:	October 16, 2013
SOLD TO:	LOS ALAMOS NATIONAL SECURITY
SHIPPING ADDRESS:	LANL FOR US DOE / NNSA SM-30 WAREHOUSE BIKINI ATOLL ROAD LOS ALAMOS, NM 87545
SHIP VIA:	**BEST WAY, FREIGHT PRE-PAID AND ADDED TO INVOICE**
CUSTOMER P.O.:	**243568**
BILLING ADDRESS:	LOS ALAMOS NATIONAL SECURITY invoices@lanl.gov
QUANTITY:	12
DESCRIPTION:	SAVY-4000 Filter Standards
SERIAL #:	243568-3L-001 thru 243568-3L-012
DATE OF MANUFACTURE:	10/13
NUMBER OF BOXES:	_____

I. POC driver code

LANL POC driver code
v03_00_02 08/06/2014

```
/**
 *  Project: On-Canister Filter Tester
 *
 *  @file OCFT_reader_v00_01_11.ino
 *  @version 0.1.11
 *  @author Joel Runnels
 *  @date 2013-08-14
 *
 *  @brief
 *  This file contains the setup function and the loop function for the
 *  OCFT driver code.
 *
 *  @details
 *  Menu-driven function added in later 0.0.x versions.
 *  The transition to version 0.1.x represents the shift to a touch-screen
 *  driven interface. Serial commands are not used in this version at all,
 *  and serial output is used only for debugging.
 *
 *  As of this version, the code is mostly functional. Major issues lie in
 *  the system volume calibration function.
 *
 *  Major improvements include:
 *  --EEPROM storage of sample pressure, sample time, and calibrated container volume
 *  --Menu driven touch screen interface
 *
 *  Basic functionality of testing a contianer volume is still working.
 *
 *  Note that there are still several debugging print lines in the
 *  code; some of them are commented out, some not.
 *
 *
 *
 *
 *
 *
 *
 *
 *  A NOTE ON LIBRARIES (read this if the code won't compile on your computer!):
 *  This program uses three external libraries provided by Adafruit for interfacing
 *  with touchscreen. These libraries are included in the "libraries" subfolder
 *  included with this package. If you do not already have these libraries installed,
 *  you need to do so, otherwise the code will not compile. To install these
 *  libraries, follow the following steps:
 *
 *  If you are using Aruino 1.0.5 or newer:
 *
```

```
 *  1) Open the Arduino IDE
 *  2) Click on "Sketch" in the menu bar
 *  3) Click on "Import Library"
 *  4) Click on "Add Library"
 *  5) You will be prompted to select the library you would like to add. Navigate to
 *     the "lib" subfolder within the OCFT folder that contains the source code for
 *     the OCFT.
 *  6) Select the first library .zip file in the list.
 *  7) Repeat steps 2-6 for the remaining libraries in the subfolder.
 *  8) Restart the Arduino IDE.
 *
 *
 *
 *  If you are using Arduino 1.0.4 or older (or if you want to do a manual install):
 *
 *  1) Locate and navigate to your Arduino sketchbook folder. For Windows, this will
 *     likely be located under "My Documents/Arduino", unless you have moved it
 *     somewhere else.
 *  2) Open the libraries folder within your sketchbook folder.
 *  3) In a separate file browser window, navigate to the OCFT code folder.
 *  4) Navigate to the "lib" subfolder in the OCFT folder.
 *  5) In the "lib" folder you should find three .zip files. Unzip all of them.
 *  6) Drag the contents of the unzipped files into the "libraries" subfolder within
 *     your sketchbook.
 *  7) If you have the Arduino IDE running, close and restart it.
 *
 *
 *  @todo Finish auto volume calibration feature, add more error handling capabilities.
 *
 **/



/************************/
/*  INCLUDE STATEMENTS  */
/************************/

#include "functions.h"
#include "define.h"
#include <TouchScreen.h>
#include "buttons.h"
#include <EEPROM.h>
#include <Adafruit_GFX.h>    // Core graphics library
#include <Adafruit_TFTLCD.h> // Hardware-specific library



/********************/
/*  SETUP FUNCTION  */
```

```
/********************/



/**
 *
 *  The setup function is the function which runs only once, at the beginning of
 *  the program. In this function, pins are designated as outputs and brought to
 *  their default state. Serial communication also commences. The system also
 *  does a check to make sure that the system's internal volume has already been
 *  calibrated.
 *
 *  @param void
 *
 *  @retval void
 *
 */

void setup ()
{
        //Set pins for relay control as outputs, set them high (off)

        pinMode(CC_101, OUTPUT);
        pinMode(CC_102, OUTPUT);
        pinMode(CC_103, OUTPUT);
    pinMode(CC_104, OUTPUT);
        pinMode(CC_105, OUTPUT);
        pinMode(CC_106, OUTPUT);

        digitalWrite(CC_101, OFF);
        digitalWrite(CC_102, OFF);
        digitalWrite(CC_103, OFF);
    digitalWrite(CC_104, OFF);
        digitalWrite(CC_105, OFF);
        digitalWrite(CC_106, OFF);


        //Begin serial communications. Ultimately serial communication will not be used in final product
        Serial.begin(9600);




        if((abs(settings_read_int_volume(1) + 1.00)) < (1.0))
    {
         Serial.print("System detects that you have never measured the internal system volume! This must be done
before any tests are performed, otherwise results will be innacurate.\r\n");

    }
```

```
    Serial.println("Serial communication is working! HA-HA!:) ");


}




/*******************/
/*  LOOP FUNCTION  */
/*******************/



/**
 *
 *  The loop function holds the code that runs repeatedly until the microcontroller
 *  is powered off. In the loop function, two objects are created; one for
 *  controlling the LCD and one for controlling the touchscreen. The LCD is reset,
 *  then communication is initialized. The function sets the proper rotation and
 *  fills the screen to white. The function then passes pointers to the touchscreen
 *  and LCD to the \ref main_menu function, at which point the logic of the \ref main_menu
 *  function takes control of the program. \ref main_menu runs until the user selects
 *  an option. After the user's requests are executed, main_menu exits and passes
 *  control back to \ref loop(), at which point loop runs again.
 *
 *  @param void
 *
 *  @retval void
 *
 */

void loop ()
{

        //Touchscreen and LCD objects are created in loop function, and passed "by reference" throughout the menus.

        Adafruit_TFTLCD tft;
        TouchScreen ts = TouchScreen(XP, YP, XM, YM, 300);
        uint16_t identifier;

        tft.reset();
        identifier = (tft).readID();
        tft.begin(identifier);
        tft.setRotation(1);
        tft.fillScreen(WHITE);
```

```
        main_menu(&tft, &ts);

}


/**
 *  Project: On-Canister Filter Tester
 *  @file buttons.cpp
 *  @version 0.1.11
 *  @author Joel Runnels
 *  @date 2013-08-14
 *
 *  @brief This file contains the button class definition
 *
 *  @details
 *  Function stores button class constructor, destructor, draw and
 *  get_status functions. Linked to buttons.h
 *
 *  @todo
 *
 */


#include "buttons.h"
#include "define.h"
#include <TouchScreen.h>
#include <Arduino.h>
#include <Adafruit_GFX.h>    // Core graphics library
#include <Adafruit_TFTLCD.h> // Hardware-specific library
#include <stdint.h>



/*************************************
/// BUTTON CLASS FUNCTION DEFINITIONS
/*************************************/


/**
 *  @brief  Button Class constructor
 *
 *  This function contains the basic button construction. It copies user
 *  input to button class variables.
 *
 *  @param  uint16_t s_x: Left edge coordinate of button
 *  @param  uint16_t s_y: Upper edge coordinate of button
 *  @param  char * s_text_string: Button text
 *  @param  size_t s_text_length: Length of button text
```

```
 *  @param  size_t text_size: Size of button text
 *  @param  Adafruit_TFTLCD * new_tft: Pointer to LCD object
 *  @param  TouchScreen * new_ts: Pointer to touchscreen object
 */

button::button(uint16_t s_x, uint16_t s_y, char * s_text_string, size_t s_text_length, size_t s_text_size, Adafruit_TFTLCD
* new_tft, TouchScreen * new_ts)
{

      //Assign input variables to the object's local values

      x = s_x;
      y = s_y;

      text_string = s_text_string;
      text_length = s_text_length;
      text_size = s_text_size;

      tft = new_tft;
      ts = new_ts;



}




/**
 *  @brief  Button Class destructor
 *
 **/

button::~button(void)
{
      ;
}




/**
 *
 *  @brief This function is responsible for displaying the button to the screen using assigned colors
 *
 *  The draw function determines the required height and width of the button
 *  to be drawn. It then draws the button using user-assigned colors, and fills in
 *  the text from the text string. It also assigns the color values to the object's
 *  local variables.
```

```
 *
 *  @param  uint16_t s_fill_color: Fill color of button
 *  @param  uint16_t s_outline_color: Outline color of button
 *  @param  uint16_t s_fill_color: Color for button text
 *
 *  @retval void
 *
 **/

void button::draw(uint16_t s_fill_color, uint16_t s_outline_color, uint16_t s_text_color)
{

      byte i = 0;

      //Assign fill color values to object variables

      fill_color = s_fill_color;
      outline_color = s_outline_color;

      text_color = s_text_color;


      //Calculate length and width of button

      x_length = (text_length * text_size * 6) + (text_size * 5);
      y_length = (text_size * 7) + (text_size * 10);


      //Draw button

      (*tft).fillRoundRect(x, y, x_length, y_length, 5, fill_color);
      (*tft).drawRoundRect(x, y, x_length, y_length, 5, outline_color);

      (*tft).setCursor(x + (text_size * 5), y + (text_size * 5));
      (*tft).setTextColor(text_color);
      (*tft).setTextSize(text_size);


      //Fill in text

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

            (*tft).write(*(text_string + i));

      }

      return;
}
```

```
/**
 *  @brief This function gets the status of the button (pressed or un-pressed) and returns
 *  it to calling function
 *
 *  This function checks whether or not the button has been pressed. If it has been,
 *  the function temporarily redraws the button with alternate colors; then draws the
 *  button back with its proper colors. Return value is dependent on status of button.
 *
 *  @retval byte: returns status of button (0 = unpressed, 1 = pressed)
 *
 **/

byte button::get_status(void)
{


      int tmp = 0;
      byte button_status = 0;
      byte i = 0;


      //Check status of button

      pinMode(13, OUTPUT);
      digitalWrite(13, HIGH);
      Point p = (*ts).getPoint();
      digitalWrite(13, LOW);
      pinMode(XM, OUTPUT);
      pinMode(YP, OUTPUT);


      //Copy object colors to temporary storage

      uint16_t tmp_fill = fill_color;
      uint16_t tmp_text = text_color;
      uint16_t tmp_outline = outline_color;


      //Map coordinates of touchscreen contact to standard pixel coordinates
      p.x = map(p.x, TS_MINX, TS_MAXX, (*tft).height(), 0);
      p.y = map(p.y, TS_MINY, TS_MAXY, (*tft).width(), 0);

      //These lines account for the fact that the screen is rotated by 90 degrees
      tmp = p.y;
      p.y = (*tft).height() - p.x;
```

```
    p.x = tmp;



    //If statement determines whether touchscreen pressure is large enough to be considered a true "touch"
    if(p.z > MINPRESSURE && p.z < MAXPRESSURE)
    {

        //Nested if statements determine whether touch is inside button coordinates or outside
        if((p.x > x) && (p.x < (x + x_length)))
        {
            if((p.y > y) && (p.y < (y + y_length)))
            {

                //redraw button with alternate colors
                draw(tmp_text, tmp_text, tmp_fill);
                button_status = 1;

                //Small delay
                delay(BUTTON_LAG_TIME);

                //Draw button with correct colors
                draw(tmp_fill, tmp_outline, tmp_text);

                //If button was pressed, function returns 1 to calling function
                return (1);

            }
        }
    }

    //If button not pressed, function returns 0 to calling function
    return(0);

}


/**
 *  Project: On-Canister Filter Tester
 *  @file buttons.h
 *  @version 0.1.11
 *  @author Joel Runnels
 *  @date 2013-08-14
 *
 *  @brief This file contains the prototype for the button class, as well as several
 *  define macros specific for operation of buttons or touchscreen.
 *
 *  @details
 *  Define macros borrowed from adafruit libraries for touchscreen and LCD interface
 *
```

```
 * @todo
 *
 */


#ifndef BUTTONS_H
#define BUTTONS_H

#include <Arduino.h>
#include <Adafruit_GFX.h>    // Core graphics library
#include <Adafruit_TFTLCD.h> // Hardware-specific library


#include <stdint.h>
#include <TouchScreen.h>



/************************/
/*   DEFINE STATEMENTS  */
/************************/

// These are the pins for the shield!
/// Define pins needed for operation of touchscreen shield
#define YP A1  // must be an analog pin, use "An" notation!
#define XM A2  // must be an analog pin, use "An" notation!
#define YM 7   // can be a digital pin
#define XP 6   // can be a digital pin

#define MINPRESSURE 10  ///< Minimum pressure threshold before a "touch" on the touchscreen is detected
#define MAXPRESSURE 10000  ///< Maximum pressure threshold above which a "touch" is not detected

#define TS_MINX 150 ///< The minimum x coordinate that will be returned from a touchscreen point
#define TS_MINY 120 ///< The minimum y coordinate that will be returned from a touchscreen point
#define TS_MAXX 920 ///< The maximum x coordinate that will be returned from a touchscreen point
#define TS_MAXY 940 ///< The maximum y coordinate that will be returned from a touchscreen point



/// Define some standard color values
#define BLACK   0x0000 ///< Black hex value
#define BLUE    0x001F ///< Blue hex value
#define RED     0xF800 ///< Red hex value
#define GREEN   0x07E0 ///< Green hex value
#define CYAN    0x07FF ///< Cyan hex value
#define MAGENTA 0xF81F ///< Magenta hex value
#define YELLOW  0xFFE0 ///< Yellow hex value
#define WHITE   0xFFFF ///< White hex value
```

```
#define BUTTON_LAG_TIME 100 ///< Time which button "waits" after being pressed



/**********************/
/*  BUTTON PROTOTYPE  */
/**********************/


/**
 *
 * @brief Basic class to support button input through touchscreen
 *
 * This class allows for basic, oval shaped buttons. It allows user to enter
 * arbitrary text with an arbitrary size, and the function automatically draws
 * the button to be the right size for the text (unless text is too large for
 * screen). The function allows for individual selection of text color, fill
 * color, and outline color. Button automatically handles alternate coloring
 * during button pressing, as long as the get_status function is being called.
 *
 * @param uint16_t x: The coordinate of the left edge of the button
 * @param uint16_t y: The coordinate of the top edge of the button
 * @param uint16_t x_length: The length of the button in the x direction
 * @param uint16_t y_length: The length of the button in the y direction
 * @param uint16_t fill_color: Default fill color of the button
 * @param uint16_t outline_color: Default outline color of the buttn
 * @param uint16_t text_color: Default text color
 * @param char * text_string: String containing button text
 * @paran size_t text_length: Length of text string
 * @param uint16_t text_size: Size of button text
 * @param Adafruit_TFTLCD * tft: Pointer to LCD object
 * @param TouchScreen * ts: Pointer to touchscreen object
 *
 *
 */
class button
{

      uint16_t x;
      uint16_t y;
      uint16_t x_length;
      uint16_t y_length;
      uint16_t fill_color;
      uint16_t outline_color;
      uint16_t text_color;
      char * text_string;
      size_t text_length;
      uint16_t text_size;
```

```
    Adafruit_TFTLCD * tft;
    TouchScreen * ts;
    //  uint16_t identifier;

public:

    button(uint16_t, uint16_t, char *, size_t, uint16_t, Adafruit_TFTLCD *, TouchScreen *);

    ~button(void);

    void draw(uint16_t, uint16_t, uint16_t);
    byte get_status(void);

};


#endif




/**
 *  Project: On-Canister Filter Tester
 *  @file define.h
 *  @version 0.1.11
 *  @author Joel Runnels
 *  @date 2013-08-14
 *
 *  @brief
 *  This file contains the #define statements used in OCFT.
 *
 *  @details
 *  This file contains various #define macros used in the other files of the
 *  OCFT driver system. Statements below are sorted by type, grouped into the
 *  following categories:
 *
 * - Computer control links
 *
 *    Contains #define macros that link P&ID codes for computer control links
 *    to specific pins on microcontroller. These pins usually control relays
 *    or other instrument control devices. These #define macros allow pins to
 *    be rearranged without replacing values in every part of code. If wires
 *    must be routed to different pins, their pin numbers can be modified in
 *    this file, and the rest of the code will function accordingly.
 *
```

```
*
* - Instrument Computer Inputs
*
*   Contains #define macros that link P&ID codes for computer indication links
*   to specific pins on microcontroller. These pins are usually analog input
*   pins (although they could also be used for SPI CS pins or I2C addresses,
*   SPI or I2C instruments are ever added). These #define macros allow pins
*   to be rearranged without replacing values in every part of code. If wires
*   must be routed to different pins, their pin numbers can be modified in
*   this file, and the rest of the code will function accordingly.
*
*
* - Instrument Identifiers
*
*   Contains #define macros that link P&ID codes for specific instruments to
*   simple integer values for use in functions. These macros are useful when
*   specific instruments need to be specified within the code. For instance,
*   the read_pressure function accepts the instrument code and pin number to
*   determine which instrument transfer function to use.
*
*
* - Instrument Control Variables
*
*   Contains #define macros that store values for instrument interfacing. These
*   macros are usually commands that are sent to an interface, and link the
*   numeric value sent to the instrument to a human-readable value.
*
*
* - EEPROM Addresses
*
*   Contains #define macros that store the EEPROM addresses for system-specific
*   values stored in non-volatile onboard EEPROM.
*
*
* - Menu Options
*
*   Contains #define macros that link menu function outputs to human-readable
*   text. The purpose of these macros is to make menu logic code more readable
*   and traceable. Further divided into groups of macros for specific menus as
*   follows:
*
*   - Main Menu
*   - Configuration Menu
*   - Diagnostic Menu
*   - Yes/No Menu
*   - Continue/Cancel Menu
*   .
*
*
*
```

```
 * - Process Function Values --
 *
 *    Contains #define macros that define set values for various process
 *    functions. Some examples include leak check runtime, number of volume
 *    calibration points, etc. Further divided into groups of macros for various
 *    function types as follows:
 *
 *    - Filter Test/Volume Measure Variables
 *    - System Volume Calibration Variables
 *    .
 *
 *
 *  @todo
 *
 **/




#ifndef DEFINE_H
#define DEFINE_H



/*****************************/
/*  Computer Control Links  */
/*****************************/
/**
 * @name Computer Control Links
 *  Contains #define macros that link P&ID codes for computer control links
 *  to specific pins on microcontroller. These pins usually control relays
 *  or other instrument control devices. These #define macros allow pins to
 *  be rearranged without replacing values in every part of code. If wires
 *  must be routed to different pins, their pin numbers can be modified in
 *  this file, and the rest of the code will function accordingly.
 *
 */
//@{
#define CC_101 51 ///< Computer control link to relay controlling VC101 (test volume relief) relay 5
#define CC_102 52 ///< Computer control link to relay controlling VC102 (test volume capture)  relay 7
#define CC_103 53 ///< Computer control link to relay controlling FPC101 (test volume pump)  relay 3
#define CC_104 50 ///< Computer control link to relay controlling FPC102 (suction head pump)  relay 2
#define CC_105 49 ///< Computer control link to relay controlling VC103 (suction head capture)  relay 6
#define CC_106 48 ///< Computer control link to relay controlling VC104 (suction head relief)  relay 4
//@}
```

```
/*******************************/
/*  Computer Indication Links  */
/*******************************/
/**
 * @anchor ci_links
 * @name Instrument Computer Inputs
 *  Contains #define macros that link P&ID codes for computer indication links
 *  to specific pins on microcontroller. These pins are usually analog input
 *  pins (although they could also be used for SPI CS pins or I2C addresses,
 *  SPI or I2C instruments are ever added). These #define macros allow pins
 *  to be rearranged without replacing values in every part of code. If wires
 *  must be routed to different pins, their pin numbers can be modified in
 *  this file, and the rest of the code will function accordingly.
 */
//@{
#define CI_101 10 ///< Computer interface to PI101 (analog read pin)
#define CI_102 11 ///< Computer interface to PI102 (analog read pin)
#define CI_103 9 ///< Computer interface to PI103 (analog read pin)
#define CI_104 12 ///< Computer interface to FI101 (analog read pin)
//@}
```

```
/*****************************/
/*  Instrument Identifiers  */
/*****************************/
/**
 * @anchor instrument_id
 * @name Instrument Identifiers
 *  Contains #define macros that link P&ID codes for specific instruments to
 *  simple integer values for use in functions. These macros are useful when
 *  specific instruments need to be specified within the code. For instance,
 *  the read_pressure function accepts the instrument code and pin number to
 *  determine which instrument transfer function to use.
 */
//@{
#define FI_101 1 ///< Mass flow indicator (measures mass flow out of sample container)

#define PI_101 1 ///< Differential pressure gauge (measures pressure drop between sample container and atmospheric)
#define PI_102 2 ///< Differential pressure gauge (measures vacuum pressure of suction head)
#define PI_103 3 ///< Absolute Pressure Gauge (measures atmospheric pressure for determining density of air)
//@}
```

```
/*********************************/
/*  Intrument Control Variables  */
/*********************************/
/**
 * @anchor instrument_control
 * @name Instrument Control Variables
 *  Contains #define macros that store values for instrument interfacing. These
 *  macros are usually commands that are sent to an interface, and link the
 *  numeric value sent to the instrument to a human-readable value.
 */
//@{
#define ON LOW ///< Pin value for turning relay on
#define OFF HIGH ///< Pin value for turning relay off
//@}
```

```
/**********************/
/*  EEPROM Addresses  */
/**********************/
/**
 * @anchor EEPROM_addresses
 * @name EEPROM Addresses
 *  Contains #define macros that store the EEPROM addresses for system-specific
 *  values stored in non-volatile onboard EEPROM.
 */
//@{
#define SAMPLE_PRESSURE_ADDRESS 20 ///< EEPROM address for sample pressure set point (starting address)
#define MAX_TIME_ADDRESS 30 ///< EEPROM address for maximum sample time
#define SUCTION_PRESSURE_ADDRESS 40 ///< EEPROM address for suction pressure set point

#define INT_VOLUME_ADDRESS 100 ///< EEPROM address for internal volume calibration address (starting address)
//@}
```

```
/******************/
/*  Menu Options  */
/******************/
/**
 * @name Menu Options
 *    Contains #define macros that link menu function outputs to human-readable
 *    text. The purpose of these macros is to make menu logic code more readable
 *    and traceable. Further divided into groups of macros for specific menus as
 *    follows:
 *    - Main Menu
 *    - Configuration Menu
 *    - Diagnostic Menu
 *    - Yes/No Menu
 *    - Continue/Cancel Menu
 */
//@{

#define CONTINUE_CANCEL 1
#define ACCEPT_REJECT 2


#define EXIT 0 ///< Value passed to calling function from menu for "Exit"



// Main Menu
#define RUN_TEST 1 ///< Value passed to calling function from menu for "Run Test"
#define CONFIGURE 2 ///< Value passed to calling function from menu for "Configure"
#define SECURE_TEST 3 ///< Value passed to calling function from menu for "Secure Test"

// Configuration Menu
#define SET_SAMPLE_PRESSURE 1 ///< Value passed to calling function from menu for "Set Sample Pressure"
#define SET_SAMPLE_TIME 2 ///< Value passed to calling function from menu for "Set Sample Time"
#define DIAGNOSTICS 3 ///< Value passed to calling function from menu for "Diagnostics"

// Secure Test Menu
#define SET_SUCTION_PRESSURE 1 ///< Value passed to calling function from menu
#define SECURE_HARDWARE 2 ///< Value passed to calling function from menu
#define RELEASE_HARDWARE 3 ///< Value passed to calling function from menu


// Diagnostic Menu
#define CALIB_SYSTEM_VOLUME 1 ///< Value passed to calling function from menu for "Calibrate System Volume"


// Yes/No Menu
#define YES (byte) 1 ///< Value passed to calling function from menu for "Yes"
```

```
#define NO (byte) 0 ///< Value passed to calling function from menu for "No"

// Continue/Cancel Menu
#define CONTINUE 1 ///< Value passed to calling function from menu for "Continue"
#define CANCEL 0 ///< Value passed to calling function from menu for "Cancel"

// Accept/Reject Menu
#define ACCEPT 1 ///< Value passed to calling function from menu for "Accept"
#define REJECT 0 ///< Value passed to calling function from menu for "Reject




//@}









/*****************************/
/*  Process Function Values  */
/*****************************/
/**
 * @name Process Function Values
 *    Contains #define macros that define set values for various process
 *    functions. Some examples include leak check runtime, number of volume
 *    calibration points, etc. Further divided into groups of macros for various
 *    function types as follows:
 *    - Filter Test/Volume Measure Variables
 *    - System Volume Calibration Variables
 */
//@{

// Filter Test/Volume Measure Variables

#define MAX_SAMPLE_PRESSURE 20 ///< Max pressure to allow user to enter for standard tests ("H2O, warning level)
#define MIN_SAMPLE_PRESSURE 2 ///< Min pressure to allow user to enter for standard tests ("H2O, warning level)
#define MAX_SUCTION_PRESSURE 250 ///< Max pressure to allow user to enter for standard tests ("H2O, warning level)
#define MIN_SUCTION_PRESSURE 10 ///< Min pressure to allow user to enter for standard tests ("H2O, warning level)
#define MAX_SAMPLE_TIME 120 ///< Max number of seconds to allow user to enter for max sample time (warning
level)
#define MIN_SAMPLE_TIME 20 ///< Minimum number of seconds to allow user to enter for max sample time (warning
level)

#define LEAK_CHECK_WAIT_TIME 10000000 ///< Number of microseconds to wait during leak-checking procedures
```

```
#define CLOGGED_FILTER_THRESHOLD 10 ///< Minimum number of cc's that must be measured to confirm non-clogged filter

#define LEAK_FREE_FLOW_RATE 1 ///< Maximum flowrate through an O-ring at which the O-ring can be considered "secure" (in scc/s). The CORRECT value is 10e-5, but this value may be adjusted during the programing process for error checking purposes.

#define LEAKY 1;
#define SECURE 0;

#define RELAY_LAG_TIME 500 ///< Number of milliseconds to wait before throwing relays in sequence (prevent microcontroller current surge)

#define SCREEN_PRINT_INTERVAL  500 ///< Number of milliseconds to wait before refreshing on-screen measurement data

// System Volume Calibration Variables

#define NUMBER_VOLUME_CALIB_POINTS 1 ///< Number of sets of volume points for system volume calibration (for averaging)
#define MINIMUM_VOLUME_CALIB_PRESSURE 3 ///< Starting pressure for each volume measurement set
#define PRESSURE_INCREMENT 3 ///< Difference between pressure values in volume measurement sets
#define NUMBER_VOLUME_CALIB_READS 1 ///< Number of volume measurement points per set of volume points


#define RERUN 1


#define TEST_VOLUME 1
#define SUCTION_VOLUME 2
#define SUCTION_HEAD_VOLUME 50.0


// Abort Codes
#define USER_ABORT 1
#define ORING_FAILED 2

//@}


#endif



/**
 * Project: On-Canister Filter Tester
 *
 * @file functions.cpp
 * @version 0.1.11
 * @author Joel Runnels
```

```
* @date 2013-08-14
*
* @brief This file contains all the function definitions for the functions
* used in OCFT driver code.
*
* @details
* Functions grouped by type, then sorted alphabetically. Functions stored by
* the following categories:
*
* - Menu Logic Functions
*
*   This category contains functions which handle flow control and logic of
*   the programming. These functions are generally the highest-level functions
*   in the program which pass user input and output to calling functions.
*
*
* - Computational Functions
*
*   This category contains functions which perform some sort of computational
*   function. These functions generally do not involve any instrument or
*   hardware interfacing, and are designed to be as general as possible.
*   Examples include trapezoidal integration function and the volume computation
*   function.
*
*
* - Hardware and Instrumentation Interface Functions
*
*   This category contains functions which interface directly with hardware
*   and instrumentation. These functions usually involve interfacing directly
*   with instrument pins, and return values of sensor readings if available.
*
* - EEPROM Interface Functions
*
*   This category contains functions which read or write to microcontroller
*   onboard EEPROM. Functions are specifically designed to read or write a
*   particular value to EEPROM. Examples include the function which reads the
*   max sample time from EEPROM.
*
* - Process Functions
*
*   This category contains functions which control higher-level processes
*   onboard the instrument. Examples of functions in this category include
*   the functions responsible for measuring volume of a container and running
*   filter tests. These functions often involve controlling hardware,
*   interfacing with instruments, and performing computations. These functions
*   usually call multiple lower-level functions during the course of their
*   execution. These functions also interface with the touchscreen to display
*   current data as needed.
*
* - Touchscreen Interface Functions
```

```
 *
 *    This category contains functions which are responsible solely for
 *    touchscreen interfacing, both displaying data to touchscreen and receiving
 *    user input from touchscreen. These functions often pass user input directly
 *    to the logic flow control functions of the software.
 *
 *
 *
 * @todo
 * Continue commenting effort, add support for vacuum suction clamping mechanism.
 *
 * Also, many functions in this file make heavy use of the millis() and micros()
 * functions for keeping track of time. These numbers will overflow (see
 * http://arduino.cc/en/Reference/Micros for more information) after a given
 * amount of time. Currently the functions do not include error handling for
 * millis() and micros() overflow. It is currently unknown what the consequences
 * of such an overflow might be on the performance of the functions. This
 * problem needs to be investigated and error handlers need to be put in place
 * where needed. micros() has an overflow time of about 70 minutes, so if the
 * microcontroller is left running longer than that erroneous results may occur.
 *
 **/




/*************************/
/*   INCLUDE STATEMENTS  */
/*************************/

#include <Arduino.h>
#include <EEPROM.h>
#include <Adafruit_GFX.h>    // Core graphics library
#include <Adafruit_TFTLCD.h> // Hardware-specific library
#include <TouchScreen.h>
#include "buttons.h"
#include "define.h"
#include "functions.h"




/*****************************/
```

```
/*   FUNCTION DEFINITIONS   */
/*****************************/









/*************************/
/*  Menu Logic Functions  */
/*************************/


/**
 * @name Menu Logic Functions
 *
 *
 *  This category contains functions which handle flow control and logic of
 *  the programming. These functions are generally the highest-level functions
 *  in the program which pass user input and output to calling functions.
 *
 *
 */

//@{




/**
 *
 *  Function contains logic of main menu and directs program to sub-menu or sub-function
 *
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int 0: returns 0 upon successful completion
 *
 */

int main_menu(Adafruit_TFTLCD * tft, TouchScreen * ts)
{

      /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-
```

```
char menu_opt = 0;



/// Function calls draw_main_menu to display menu options to user and pass user selection to function
menu_opt = draw_main_menu(3, 3, tft, ts);


/// Switch case based on user input
switch (menu_opt)
{
case RUN_TEST:
{


    /// Display confirmation menu to user, give user option to cancel run

    char run_header [] = "Run Test";
    char run_message [] = "Please make sure mask is\r\n placed securely over\r\n canister. When mask is\r\n
secure, press\r\n \"Continue\".";

    menu_opt = draw_continue_cancel(2, 2, run_header, sizeof(run_header), run_message, sizeof(run_message),
CONTINUE_CANCEL, tft, ts);


    /// If user selects continue, run filter test
    if(menu_opt == CONTINUE)
    {

        run_test_ts(tft, ts);

    }

    break;

}


case CONFIGURE:

    /// If user selects configure option, display configure menu
    configure_menu(tft, ts);
    break;


case SECURE_TEST:
```

```
            /// If user selects secure test option, display secure test menu
            secure_test_menu(tft, ts);
            break;
        }
        return (0);


        }
```

```
/**
 *
 *  Function contains logic of configure menu and passes control to appropriate sub-menus
 *
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int 0: returns 0 upon successful completion
 *
 */

int configure_menu(Adafruit_TFTLCD * tft, TouchScreen * ts)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    char menu_opt = 0;
    char repeat = 1;

    /// Main logic sits inside a repeat loop. Menu logic repeats until user exits menu.
    while (repeat)
    {
```

```
/// Displays menu to user through LCD, returns user selection to menu_opt
menu_opt = draw_configure_menu(3, 2, tft, ts);


/// Switch statement based on menu_opt
switch (menu_opt)
{



    /// This case executes is user selects to set sample pressure
case SET_SAMPLE_PRESSURE:
{

    ///Read current max_pressure from EEPROM
    double max_pressure = settings_read_pressure();
    repeat = 1;

    /// Rest of case resides in a while loop for convenient repeating as needed.
    while(repeat)
    {

        {


            char sample_pressure_header [] = "Sample pressure drop";
            char in_water [] = "\"H2O";

            /// Input to the scroll_menu function was so complicated that I decided to lump it
            /// all into one structure for increased code readability.
            scroll_menu_input pressure_menu;

            pressure_menu.header_size = 2;
            pressure_menu.button_size = 3;
            pressure_menu.scroll_title = sample_pressure_header;
            pressure_menu.scroll_size = sizeof(sample_pressure_header);
            pressure_menu.units = in_water;
            pressure_menu.unit_size = sizeof(in_water);
            pressure_menu.val_init = max_pressure;
            pressure_menu.val_min = 0;
            pressure_menu.val_max = 255;
            pressure_menu.warning_min = 2;
            pressure_menu.warning_max = MAX_SAMPLE_PRESSURE;
            pressure_menu.increment = 0.5;


            /// Draws scroll menu which allows user to enter max pressure drop. Returns entered value
            /// from menu to max_pressure variable.
```

```
                    max_pressure = draw_scroll_menu(pressure_menu, tft, ts);

                }


                /// If the user entered a pressure considered to be too large, then the program displays a warning
                if((max_pressure > MAX_SAMPLE_PRESSURE) || (max_pressure < MIN_SAMPLE_PRESSURE))
                {


                    char sample_pressure_header [] = "Pressure Error";
                    char sample_pressure_error [] = "You have entered a sample pressure which is outside of the
recommended range  for this system. Would\r\n you like to enter a\r\n different sample\r\n pressure?";

                    menu_opt = draw_yes_no_menu (3, 2, sample_pressure_header,
sizeof(sample_pressure_header), sample_pressure_error, sizeof(sample_pressure_error), tft, ts);


                    /// If user acknowledges warning message and decides to change pressure, loop repeats.
                    if(menu_opt == YES)
                    {
                        repeat = 1;

                    }

                    /// If user ignores warning, the repeat variable is set to zero and the pressure entered
                    /// by the user is written to EEPROM
                    else
                    {
                        settings_write_pressure(max_pressure);
                        repeat = 0;
                    }

                }

                /// If user enters a pressure within the system specs, no warning is given.
                else
                {
                    settings_write_pressure(max_pressure);
                    repeat = 0;
                }

            }

            /// Once out of sub-while loop, set repeat back to 1 for repeating of main menu
            repeat = 1;

            break;

        }
```

```
/// This case executes if user selects to set the time
case SET_SAMPLE_TIME:
{
    /// Read current sample time from EEPROM and assign to sample_time
    byte sample_time = settings_read_time();
    repeat = 1;

    /// Logic sits in nested while loop for easy repeating
    while (repeat)
    {

        char sample_time_header [] = "Max sample time";
        char seconds [] = "seconds";

        // As noted above, clumped these variables into a struct to increase readability.
        scroll_menu_input time_menu;

        time_menu.header_size = 2;
        time_menu.button_size = 3;
        time_menu.scroll_title = sample_time_header;
        time_menu.scroll_size = sizeof(sample_time_header);
        time_menu.units = seconds;
        time_menu.unit_size = sizeof(seconds);
        time_menu.val_init = sample_time;
        time_menu.val_min = 0;
        time_menu.val_max = 255;
        time_menu.warning_min = 2;
        time_menu.warning_max = MAX_SAMPLE_TIME;
        time_menu.increment = 10;



        /// Draw scroll menu for user to enter new sample time
        sample_time = draw_scroll_menu(time_menu, tft, ts);


        /// If user enters sample time outside of recommended range of system, system enters conditional
        /// to display warning to user
        if((sample_time > MAX_SAMPLE_TIME) || (sample_time < MIN_SAMPLE_TIME))
        {

            char sample_time_header [] = "Sample Time Error";
            char sample_time_error [] = "You have entered a sample time which is outside of  the
recommended range for this system. Would you\r\n like to enter a different sample time?";
```

```
                    /// Display warning to user to ask whether they would like to change their entered time to
                    /// a time within system limits.
                    menu_opt = draw_yes_no_menu (3, 2, sample_time_header, sizeof(sample_time_header),
sample_time_error, sizeof(sample_time_error),  tft, ts);

                    /// If user selects to enter new time, value is not written to EEPROM and sub while loop repeats
                    if(menu_opt == YES)
                    {
                        repeat = 1;
                    }

                    /// If user selects to ignore warning, sample time is written to EEPROM and loop exits.
                    else
                    {
                        settings_write_time(sample_time);
                        repeat = 0;
                    }
                }

                /// If user enters valid time, no warning is given and value is simply written to EEPROM.
                else
                {
                    settings_write_time(sample_time);
                    repeat = 0;
                }
            }


            repeat = 1;

            break;

        }



        /// This case executes if user selects diagnostics
        case DIAGNOSTICS:

            /// Display diagnostics menu to user, repeat configuration menu upon exiting.
            diagnostics_menu(tft, ts);

            repeat = 1;

            break;

        /// This case executes if user selects to exit menu
        case EXIT:
```

```
                /// Exit configuration menu, return back to main menu
                repeat = 0;

                break;
            }
        }

        return (0);

}
```

```
/**
 *
 *  Function contains logic of the SECURE TEST menu and passes control to appropriate sub-menus
 *
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int 0: returns 0 upon successful completion
 *
 */

int secure_test_menu(Adafruit_TFTLCD * tft, TouchScreen * ts)
{
```

```
/// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

char menu_opt = 0;
char repeat = 1;

/// Main logic sits inside a repeat loop. Menu logic repeats until user exits menu.
while (repeat)
{


    /// Displays menu to user through LCD, returns user selection to menu_opt
    menu_opt = draw_secure_test_menu(3, 2, tft, ts);


    /// Switch statement based on menu_opt
    switch (menu_opt)
    {



        /// This case executes is user selects to set SUCTION pressure
    case SET_SUCTION_PRESSURE:
    {

        ///Read current max_pressure from EEPROM
        double max_pressure1 = settings_read_pressure1();
        repeat = 1;

        /// Rest of case resides in a while loop for convenient repeating as needed.
        while(repeat)
        {

            {


                char suction_pressure_header [] = "Suction pressure ";
                char in_water [] = "\"H2O";

                /// Input to the scroll_menu function was so complicated that I decided to lump it
                /// all into one structure for increased code readability.
                scroll_menu_input pressure_menu;

                pressure_menu.header_size = 2;
                pressure_menu.button_size = 3;
                pressure_menu.scroll_title = suction_pressure_header;
                pressure_menu.scroll_size = sizeof(suction_pressure_header);
                pressure_menu.units = in_water;
                pressure_menu.unit_size = sizeof(in_water);
                pressure_menu.val_init = max_pressure1;
                pressure_menu.val_min = 0;
```

```
            pressure_menu.val_max = 350;
            pressure_menu.warning_min = MIN_SUCTION_PRESSURE;
            pressure_menu.warning_max = MAX_SUCTION_PRESSURE;
            pressure_menu.increment = 10;


            /// Draws scroll menu which allows user to enter max pressure drop. Returns entered value
            /// from menu to max_pressure variable.
            max_pressure1 = draw_scroll_menu(pressure_menu, tft, ts);

        }


        /// If the user entered a pressure considered to be too large, then the program displays a warning
        if((max_pressure1 > MAX_SUCTION_PRESSURE) || (max_pressure1 < MIN_SUCTION_PRESSURE))
        {


            char sample_pressure_header [] = "Pressure Error";
            char sample_pressure_error [] = "You have entered a suction pressure which is outside of the
recommended range  for this system. Would\r\n you like to enter a\r\n different suction\r\n pressure?";

            menu_opt = draw_yes_no_menu (3, 2, sample_pressure_header,
sizeof(sample_pressure_header), sample_pressure_error, sizeof(sample_pressure_error), tft, ts);


            /// If user acknowledges warning message and decides to change pressure, loop repeats.
            if(menu_opt == YES)
            {
                repeat = 1;

            }

            /// If user ignores warning, the repeat variable is set to zero and the pressure entered
            /// by the user is written to EEPROM
            else
            {
                settings_write_pressure1(max_pressure1);
                repeat = 0;
            }

        }

        /// If user enters a pressure within the system specs, no warning is given.
        else
        {
            settings_write_pressure1(max_pressure1);
            repeat = 0;
        }
```

```
                }

                /// Once out of sub-while loop, set repeat back to 1 for repeating of main menu
                repeat = 1;

                break;
            }




            /// secures hardware
             case SECURE_HARDWARE:
        {


            /// Display confirmation menu to user, give user option to cancel run

            char run_header [] = "Secure Hardware";
            char run_message [] = "Please make sure both o-rings are in place, and hardware fits securely over\r\n
canister." ;

            menu_opt = draw_continue_cancel(2, 2, run_header, sizeof(run_header), run_message, sizeof(run_message),
CONTINUE_CANCEL, tft, ts);


            /// If user selects continue, secure hardware
            if(menu_opt == CONTINUE)
            {

              run_secure_hardware_ts(tft, ts);

            }

            break;

        }

        ///releases hardware.
        ///****NEED TO ADD SECONDARY LEAK CHECK TO CONFIRM SEAL DURING ALL TESTS!!!!!
        case RELEASE_HARDWARE:{


         char run_header [] = "Release Hardware";
            char run_message [] = "Would you like to release the hardware, and finish testing?" ;

            menu_opt = draw_continue_cancel(2, 2, run_header, sizeof(run_header), run_message, sizeof(run_message),
CONTINUE_CANCEL, tft, ts);
```

```
            /// If user selects continue, secure hardware
            if(menu_opt == CONTINUE)
            {

              run_release_hardware_ts(tft, ts);

            }

            break;

        }
         /// This case executes if user selects to exit menu
            case EXIT:

                /// Exit configuration menu, return back to main menu
                repeat = 0;

                break;
            }


            }

}
```

```
/**
 *
 *  This function contains the logic for the diagnostics sub-menu. Note that
 *  this menu will need to be expanded upon the addition of more diagnostic tools.
 *
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int 0: returns 0 upon successful completion
 *
 */


int diagnostics_menu(Adafruit_TFTLCD * tft, TouchScreen * ts)
```

```
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    char menu_opt = 0;
    byte repeat = 1;
    byte i = 0;
    byte leak;
    double system_volume = 0.0;
    double avg_system_volume = 0.0;



    /// Menu logic in while loop. Loop repeats until user decides to exit menu.
    while (repeat)
    {


        /// Display graphical diagnostic menu to user, pass user input to menu_opt
        menu_opt = draw_diagnostic_menu(3, 2, tft, ts);


        /// Switch based on user input
        switch (menu_opt)
        {

        /// This case executes if user selects to calibrate system volume
        case CALIB_SYSTEM_VOLUME:
        {


            /// If user selects to recalibrate system volume, software displays message asking user to confirm
            /// action and to cover system inlet

            char calib_header [] = "Volume Calibration";
            char calib_message [] = "You have chosen to\r\n recalibrate the system\r\n volume. To proceed,\r\n
please cover the system\r\n sample inlet securely\r\n so that no air can enter\r\n the system test volume.\r\n When
system is secure,\r\n press \"Continue\".";

            /// Pass user input to menu_opt
            menu_opt = draw_continue_cancel(2, 2, calib_header, sizeof(calib_header), calib_message,
sizeof(calib_message), CONTINUE_CANCEL, tft, ts);



            /// If user selects to continue with volume calib, run the calib_system_volume_ts
            if(menu_opt == CONTINUE)
            {

                calib_system_volume_ts(tft, ts);
```

```
                }

                /// Repeat menu upon completion
                repeat = 1;

                break;

            }



            /// This case executes if user selects to exit menu
            case EXIT:
                /// If user selects to exit diagnostic menu, repeat is set to zero, menu exits.
                repeat = 0;
                break;

            }

        }

        return (0);

}




//@}









/*****************************/
/*  Computational Functions  */
/*****************************/
```

```
/**
 * @anchor comp_functions
 * @name Computational Functions
 *
 *  This category contains functions which perform some sort of computational
 *  function. These functions generally do not involve any instrument or
 *  hardware interfacing, and are designed to be as general as possible.
 *  Examples include trapezoidal integration function and the volume computation
 *  function.
 *
 */

//@{




/**
 *
 *  Function computes mass flow into a container given pressure change, container volume, and starting density
 *
 *  @param double P1: The initial pressure of container (inches of water, absolute pressure)
 *  @param double dP: Pressure drop during the course of the run (inches of water, differential pressure)
 *  @param double V: Volume of container (cubic centimeters)
 *  @param double rho: density of air inside container (NOT the same as density
 *  of air at current atmospheric pressure, nescessarily. Can be derived from
 *  P1/P_stp. (dimensionless, normalized to STP density)
 *
 *  @retval double volume, returns computed volume of container
 *
 */


double compute_mass_flow_in(double P1, double dP, double V, double rho)
{

      /// See derivation for computation of volume on page 57 of OCFT lab book

      return((V*rho*dP)/P1);

}
```

```
/**
 *
 *  Function computes volume of container given pressure drop, mass flow out, and density
 *
 *  @param double P1: The initial pressure of container (inches of water, absolute pressure)
 *  @param double dP: Pressure drop during the course of the run (inches of water, differential pressure)
 *  @param double dm: Mass flow out of container (standard cubic centimeters, positive = mass flow OUT of container)
 *  @param double rho: density of air (dimensionless, normalized to STP density)
 *
 *  @retval double volume, returns computed volume of container
 *
 */


double compute_volume(double P1, double dP, double dm, double rho)
{

      /// See derivation for computation of volume on page 42 of OCFT lab book

      return((dm*P1)/(dP*rho));

}
```

```
/**
 *
 *  Performs a trapezoidal integration of x = f(t) from t to t+dt.
 *
 *  @param double dx_current: dx/dt(t+dt)
 *  @param double dx_last: dx/dt(t)
 *  @param double dt: time (in seconds)
 *
 *  @retval double: Trapezoidal integral
 *
 */


double trap_integrate(double dx_current, double dx_last, double dt)
{
```

```
	return (((dx_current + dx_last)*(dt/2.0)));

}









/**
 *
 *  Computes the average derivative of x = f(t) from t to t + dt.
 *
 *  @param double x_current: x(t + dt)
 *  @param double x_last: x(t)
 *  @param double dt: time (in seconds)
 *
 *  @retval double: Average derivative
 *
 */

double derivative(double x_current, double x_last, double dt)
{

	return ((x_current - x_last)/dt);

}



//@}









/***********************************************/
/*  Hardware and Instrument Interface Functions  */
/***********************************************/
```

```
/**
 * @anchor instrument_interface_functions
 * @name Hardware and Instrument Interface Functions
 *
 *
 *  This category contains functions which interface directly with hardware
 *  and instrumentation. These functions usually involve interfacing directly
 *  with instrument pins, and return values of sensor readings if available.
 *
 */

//@{




/**
 *
 *  This function reads mass flow through Honeywell HAFBLF0750CAAX5 mass flow meter
 *  The derivation of the transfer function for this meter can be found on page 38 of OCFT logbook 1
 *
 *  @param int mass_pin: Analog read pin to which v_out of the mass flow meter is connected
 *  @param char sensor: Type of sensor being read. See \ref define.h for more details. Note
 *  that currently only one mass flow sensor is supported, so the sensor variable is not
 *  used in the function.
 *
 *  @retval double flow: returns the flow through the mass flow meter (in scc/s)
 *
 *
 *
 */


double read_mass_flow(int mass_pin, char sensor)
{


    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-
    double flow;

    ///Analog read of v_out from mass flow meter
    int sensor_value = 0.0;



    sensor_value = analogRead(mass_pin);
```

```
    ///Calculate flow according to transfer function
    flow = ((1875.0*((sensor_value/1023.0)-0.5))/60.0);



    ///  In tests, output vs. actual flow rate was found to vary slightly
    ///  between positive and negative values. This if statement accounts
    ///  for that discrepancy. Note that these terms are specific to this
    ///  individual flow meter. Ultimately a calibration function should
    ///  be included.
    if(flow > 0)
    {
        flow = (flow * 1.1224) + 0.8338;
    }
    else
    {
        flow = (flow * 1.086) + 1.8761;
    }

    return(flow);

}
```

```
/**
 *
 *  This function reads pressure off of pressure sensors in the OCFT system.
 *  Currently, the function includes the capability to read from two
 *  different sensors: Honeywell SSCDRRN020NDAA5, and Honeywell SSCDANN030PAAA5.
 *
 *  @param int pressure_pin: Analog pin to which v_out of the pressure sensor is connected
 *  @param char sensor: Type of sensor being read. See \ref define.h for more details
 *
 *  @retval double pressure: pressure read from sensor (in "H2O)
 *
 *  \ingroup hardware_instrument_function_def
 *
 */
```

```
double read_pressure(int pressure_pin, char sensor, byte rolling_average)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-
    double pressure = 0.0;

    double current_pressure = 0;

    int i = 0;

    /// Analog read of v_out from pressure sensor
    int sensor_value = 0;

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

        sensor_value = analogRead(pressure_pin);



        /// Select transfer function based on sensor type
        switch (sensor)
        {
        case PI_101:
            /// Pressure transfer function derived on page 34 of OCFT logbook 1
            pressure = pressure + ((((sensor_value/1023.0)-0.1)*(50.0)) - 20.0)*-1;
            break;

        case PI_102:
            /// Pressure transfer function derived on page 73 of OCFT logbook 1
            /// Note conversion factor of 27.67... to convert from PSI to "H2O

            pressure = pressure + (((((sensor_value/1024.0)-0.1)*(37.5)) - 15.0)*(-27.6704523));



            break;

        case PI_103:
            /// Pressure transfer function derived on page 45 of OCFT logbook 1
            /// Note conversion factor of 27.67... to convert from PSI to "H2O
            pressure = pressure + (sensor_value - (102.3))*(30.0/(818.4)) * 27.6704523;
            break;


        }

        if (rolling_average > 1)
```

```
            {

                  delay(1);

            }

      }

      if (rolling_average > 1)
      {

            pressure = pressure/double(i);

      }

      return(pressure);

}


//@}
```

```
/*********************************/
/*  EEPROM Interface Functions  */
/*********************************/
```

```
/**
 * @name EEPROM Interface Functions
 *
 *  This category contains functions which read or write to microcontroller
```

```
 *  onboard EEPROM. Functions are specifically designed to read or write a
 *  particular value to EEPROM. Examples include the function which reads the
 *  max sample time from EEPROM.
 *
 *
 */

//@{



/**
 *
 *  This function interfaces with the microcontroller's EEPROM to read the maximum
 *  sample read time set by users.
 *
 *  @param void
 *
 *  @retval byte maximum sample time
 *
 */

byte settings_read_time(void)
{

      return (EEPROM.read(MAX_TIME_ADDRESS));

}









/**
 *
 *  This function interfaces with the microcontroller onboard EEPROM memory to
 *  write the internal system volume calibration to be stored in memory. Function
 *  writes an array of volume data with corresponding pressure drops.
 *
 *  @param pv_data * data_array: pointer to an array of pressure-volume data array
 *
 *  @retval byte 0: returns 0 upon completion
 *
 */
```

```
byte settings_write_int_volume(pv_data * data_array)
{


    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-
    byte digits [6];

    double volume;
    float power_of_ten = 1000;

    int i = 0;
    int j = 0;



    ///For loop iterates through array of pressure/volume data
    for(j = 0; j < NUMBER_VOLUME_CALIB_POINTS; j++)
    {

        ///Pulls current volume value out of array and stores it in the volume variable
        volume = ((*(data_array + j)).volume);


        ///For loop splits volume into 6 bytes with each byte representing one decimal place of the value
        for(i = 0; i < 6; i++)
        {
            digits [i] = volume/power_of_ten;

            volume = volume - (digits[i] * power_of_ten);

            EEPROM.write((INT_VOLUME_ADDRESS + i + (j*10)), digits [i]);

            power_of_ten = power_of_ten/10.0;

        }


        power_of_ten = 1000;

        ///Writes a zero to the next byte after volume information to indicate to parent function that volume has been
stored.
        EEPROM.write((INT_VOLUME_ADDRESS + i + (j*10)), 0);

        ///Writes corresponding pressure information to EEPROM
        EEPROM.write((INT_VOLUME_ADDRESS + i + (j*10) + 1), (*(data_array + j)).pressure);

    }

    return (0);
```

```
}
```

```
/**
 *
 *  Function writes user-determined maximum pressure drop for standard runs to microcontroller onboard EEPROM
 *
 *  @param double max_pressure: Maximum pressure for standard volume reads
 *
 *  @retval byte 0: returns 0 upon completion
 *
 */

byte settings_write_pressure(double max_pressure)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    byte integers;
    byte decimals;


    /// Pull current value of pressure from EEPROM
    double current_val = EEPROM.read(SAMPLE_PRESSURE_ADDRESS) + ((EEPROM.read(SAMPLE_PRESSURE_ADDRESS
+ 1))/100.0);


    /// If current value is different from the new value, write the new value to EEPROM
    if(abs(current_val - max_pressure) > 0.1)
    {

        /// Value stored in two bytes of EEPROM; one byte for integer (>1) and one byte for decimal places

        integers = max_pressure/1;
```

```
        EEPROM.write(SAMPLE_PRESSURE_ADDRESS, integers);

        decimals = (int)((max_pressure - integers) * 100.0);

        EEPROM.write(SAMPLE_PRESSURE_ADDRESS + 1, decimals);

    }

    return (0);

}



/**
 *
 *  Function writes user-determined SUCTION pressure drop for standard runs to microcontroller onboard EEPROM
 *
 *  @param double max_pressure: Maximum pressure for standard volume reads
 *
 *  @retval byte 0: returns 0 upon completion
 *
 */

byte settings_write_pressure1(double max_pressure1)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    unsigned char write_val = 0;
    unsigned char current_val = 0;

    /// Pull current value of pressure from EEPROM
    current_val = EEPROM.read(SUCTION_PRESSURE_ADDRESS);

    write_val = max_pressure1/10;


    /// If current value is different from the new value, write the new value to EEPROM
    if(write_val != current_val)
    {

        /// Value stored in two bytes of EEPROM; one byte for integer (>1) and one byte for decimal places

        EEPROM.write(SUCTION_PRESSURE_ADDRESS, write_val);

    }

    return (0);
```

```
}
```

```
/**
 *
 *  Function writes maximum time for standard sample to microcontroller EEPROM
 *
 *  @param byte max_time: Maximum time for a standard read (in seconds, max 255)
 *
 *  @retval byte 0: returns 0 upon completion
 *
 */

byte settings_write_time(byte max_time)
{
      /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

      /// Reads current value of max time from EEPROM
      byte current_val = EEPROM.read(MAX_TIME_ADDRESS);

      /// Compares current value to new value. If they are different, function writes new time.
      if(current_val != max_time)
      {
            EEPROM.write(MAX_TIME_ADDRESS, max_time);
      }

      return (0);

}
```

```
/**
 *
 *  Function reads maximum pressure for a standard run from EEPROM
 *
 *  @param void
 *
 *  @retval  double pressure: returns pressure reading
 *
 */

double settings_read_pressure(void)
{
      /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

      double pressure = 0;

      /// Value stored in two bytes of EEPROM; one byte for integer (>1) and one byte for decimal places

      pressure = EEPROM.read(SAMPLE_PRESSURE_ADDRESS);

      pressure = pressure + ((EEPROM.read(SAMPLE_PRESSURE_ADDRESS + 1))/100.0);

      return (pressure);

}




/**
 *
 *  Function reads maximum SUCTION pressure for a standard run from EEPROM
 *
 *  @param void
 *
 *  @retval  double pressure: returns pressure reading
 *
 */

double settings_read_pressure1(void)
{
      /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

      double pressure1 = 0;

      /// Value stored in two bytes of EEPROM; one byte for integer (>1) and one byte for decimal places

      pressure1 = EEPROM.read(SUCTION_PRESSURE_ADDRESS)*10;
```

```
    return (pressure1);

}
```

```
/**
 *
 *  Function reads internal volume of system. Function returns volume as a function
 *  of pressure drop (to account for the fact that the system includes some flexible
 *  tubing, and returns a higher volume for greater pressure drops). Function reads
 *  pressure, and performs linear interpolation between stored volume data points to
 *  get most accurate volume.
 *
 *  @param double pressure: pressure to search for in pressure vs. volume data
 *
 *  @retval double volume: returns linearly interpolated volume for given pressure drop
 *
 */

double settings_read_int_volume(double pressure)
{
    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    double volume = 0;
    double slope = 0;
    double v_last = 0;

    float power_of_ten = 1000;

    byte data_pressure;
    int i = 0;
    int j = 0;


    pv_data data_array [NUMBER_VOLUME_CALIB_POINTS];


    /// Simple error check. If the system volume has been written to, then condition value will
    /// be zero, so condition will be false
```

```
if(EEPROM.read(INT_VOLUME_ADDRESS + 6))
{
    return(-1);
}



/// For loop runs through the volume data points stored in EEPROM and stores them in a temporary
/// array for easier manipulation.
for(j = 0; j < NUMBER_VOLUME_CALIB_POINTS; j++)
{
    for(i = 0; i < 6; i++)
    {

        volume = volume + (EEPROM.read((INT_VOLUME_ADDRESS + i + (j*10))) * power_of_ten);

        power_of_ten = power_of_ten/10.0;


    }


    (*(data_array + j)).volume = volume;
    (*(data_array + j)).pressure = EEPROM.read(INT_VOLUME_ADDRESS + 7 + (j*10));

    power_of_ten = 1000;
    volume = 0;

}

j = 0;


/// While loop sorts through newly created pressure/volume data array until it finds a
/// pressure greater than the input pressure, or until it reaches the end of the array.
while(((pressure > (*(data_array + j)).pressure)) && (j < (NUMBER_VOLUME_CALIB_POINTS - 1)))
{
    j++;

    /// If the pressure entered is the same as a pressure stored in the data array, function
    /// returns the corresponding volume value
    if(pressure == ((*(data_array + j)).pressure))
    {
        return ((*(data_array + j)).volume);
    }


}
```

```
    /// If the pressure entered is lower than the lowest pressure in the array, then the function
    /// returns the volume corresponding to the lowest logged pressure drop.
    if(j == 0)
    {

        return ((*(data_array + 0)).volume);

    }


    /// If the pressure entered is higher than the highest pressure in the array, the function
    /// returns the volume corresponding to the highest pressure drop.
    if (j == (NUMBER_VOLUME_CALIB_POINTS - 1))
    {
        return ((*(data_array + j)).volume);
    }


    /// Assuming that the pressure drop is between the minimum and maximum pressure values in
    /// the array and not exactly the same as a pressure drop in the array, function performs a
    /// linear interpolation to get most accurate volume.

    slope = (((*(data_array + j)).volume) - ((*(data_array + j - 1)).volume))/(((*(data_array + j)).pressure) - ((*(data_array
+ j - 1)).pressure));

    volume = ((*(data_array + j-1)).volume) + ((pressure - ((*(data_array + j - 1)).pressure)) * slope);



    return (volume);

}


//@}
```

```
/***********************/
/*  Process Functions  */
/***********************/



/**
 * @name Process Functions
 *
 *  This category contains functions which control higher-level processes
 *  onboard the instrument. Examples of functions in this category include
 *  the functions responsible for measuring volume of a container and running
 *  filter tests. These functions often involve controlling hardware,
 *  interfacing with instruments, and performing computations. These functions
 *  usually call multiple lower-level functions during the course of their
 *  execution. These functions also interface with the touchscreen to display
 *  current data as needed.
 *
 *
 */


//@{




/**
 *
 *  This function measures and calibrates the system volume for a range of pressure drops,
 *  determined by the parameters set in define.h. The function performs several volume
 *  reads for pre-determined pressure drops, and logs each data point in a 2D array of
 *  pv_data structures. It then takes the average volume for each pressure range (if each
 *  pressure range was read more than once). It then displays the data collected to the
 *  user for approval. Upon approval from user, function logs data to onboard EEPROM memory.
 *
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int 0: returns 0 upon successful completion
 *
 *  @todo Finish writing function. So far, the actual volume reading is working just fine,
 *  but the data display has yet to be written. I would like to add a graphing feature that
 *  graphs the data to the user, but it remains to be seen whether or not I have time to do so.
 */
```

```
int calib_system_volume_ts(Adafruit_TFTLCD * tft, TouchScreen * ts)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-
      /// Create test variables
    double system_volume = 0;
    byte leak = 0;
    byte clogged = 0;
    byte abort_status = 0;



    char abort_button_text [] = "Abort";
    char done_button_text [] = "Done";

    byte i = 0;

    double suction_pressure = 0;




    /// Create and draw abort button
    button abort_button((*tft).width() - 90, (*tft).height() - 40, abort_button_text, sizeof(abort_button_text), 2, tft, ts);



    /// Display secure hardware header to LCD
    (*tft).fillScreen(WHITE);

    (*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(3);
    (*tft).print("Filter Test");

    (*tft).setCursor(0, 50);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print(" Securing hardware to lid");


    (*tft).setCursor(0, 90);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print("  Sample P:       \"H2O");
```

```
    (*tft).setCursor(0, 120);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);\
    (*tft).print(" Suction P:       \"H2O");

    abort_button.draw(WHITE, BLACK, RED);


    suction_pressure = settings_read_pressure1();

    secure_hardware_ts(&abort_status, tft, &abort_button, &suction_pressure);



    //*******

//      char abort_button_text [] = "Abort";
//      char done_button_text [] = "Done";

//      byte i = 0;
    byte j = 0;


    /// Draw calibration information to LCD

    (*tft).fillScreen(WHITE);

    (*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(3);
    (*tft).print("System Volume");

    (*tft).setCursor(0, 50);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print(" Calibration in progress..");

    (*tft).setCursor(0, 90);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print(" Delta P:       \"H2O");



    (*tft).setCursor(0, 120);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);\
    (*tft).print("      dm:       scc/s");
```

```
    (*tft).setCursor(0, 150);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print(" Delta M:       scc");

    (*tft).setCursor(0, 180);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print("  Volume:       cc");

    (*tft).setCursor(0, 210);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print(" Iteration: ");




    /// Create abort button object in lower corner of screen
//      button abort_button((*tft).width() - 90, (*tft).height() - 40, abort_button_text, sizeof(abort_button_text), 2, tft,
ts);

    abort_button.draw(WHITE, BLACK, RED);

    (*tft).setTextColor(BLACK);


//      byte abort_status = 0;
//      byte clogged = 0;
//      byte leak = 0;
    int num_runs = 0;

    double pressure_input = 0;
//      double system_volume = 0;
    double volume = 0;

    /// Create two pv_data arrays, one for average data and one to log each individual data point
    pv_data data_array [NUMBER_VOLUME_CALIB_READS][NUMBER_VOLUME_CALIB_POINTS];
    pv_data avg_data_array [NUMBER_VOLUME_CALIB_POINTS];



    /// Fill average array with zeroes
    for(j = 0; j < NUMBER_VOLUME_CALIB_POINTS; j++)
    {
        (*(avg_data_array + i)).pressure = 0;
        (*(avg_data_array + i)).volume = 0;
```

```
    }


    /// This for loop repeats each set of reads as specified in define.h
    for(j = 0; j < NUMBER_VOLUME_CALIB_READS; j++)
    {

        /// This for loop cycles through each pressure drop level specified in define.h
        for(i = 0; i < NUMBER_VOLUME_CALIB_POINTS; i++)
        {

            volume = 0;
            pressure_input = 0;

            num_runs = num_runs + 1;

            /// Draw blank spot on LCD for new data, update iteration number
            (*tft).fillRect(130, 210, 100, 15, WHITE);
            (*tft).setCursor(0, 210);
            (*tft).print("          ");
            (*tft).print(num_runs);
            (*tft).print("/");
            (*tft).print(NUMBER_VOLUME_CALIB_POINTS * NUMBER_VOLUME_CALIB_READS);

            /// Get next pressure drop level
            pressure_input = ((i * PRESSURE_INCREMENT) + MINIMUM_VOLUME_CALIB_PRESSURE);



            /// Measure system volume at specified pressure drop
double q_leak=0;
            if (num_runs == NUMBER_VOLUME_CALIB_POINTS * NUMBER_VOLUME_CALIB_READS)
            {

                measure_volume_ts(pressure_input, settings_read_time(), 0, &volume, &leak, &abort_status, YES, tft,
&abort_button, &q_leak);

            }
            else
            {

                measure_volume_ts(pressure_input, settings_read_time(), 0, &volume, &leak, &abort_status, NO, tft,
&abort_button, &q_leak);

            }

            /// Assign volume measured to next spot in data_array
            data_array[j][i].volume = volume;

            /// Assign current pressure drop to next spot in data_array
```

```
            data_array[j][i].pressure = pressure_input;




            /// Reset volume and pressure_input to zero
            volume = 0;
            pressure_input = 0;


            /// Add current readings to correct slots in the avg_data_array
            avg_data_array[i].pressure = avg_data_array[i].pressure + data_array[j][i].pressure;

            avg_data_array[i].volume = avg_data_array[i].volume + data_array[j][i].volume;

            /// If the read has been aborted, then exit nested for loops
            if(abort_status || leak)
            {
                  i = NUMBER_VOLUME_CALIB_POINTS;
                  j = NUMBER_VOLUME_CALIB_READS;
            }

            delay(1000);

      }
}




/// If the calibration has not been aborted, then draw data on screen to user.
/// (Note: this is where there is a lot of work still to be done!)
if(!abort_status && !leak)
{

      /// Take the average of all the readings collected
      for(j = 0; j < NUMBER_VOLUME_CALIB_READS; j++)
      {
            (*(avg_data_array + j)).pressure = (*(avg_data_array + j)).pressure / NUMBER_VOLUME_CALIB_READS;
            (*(avg_data_array + j)).volume = (*(avg_data_array + j)).volume / NUMBER_VOLUME_CALIB_READS;
      }


      /// Create variables used in drawing data to screen
      double max_pressure = 0;
      double min_pressure = 10000;
      double max_volume = 0;
      double min_volume = 10000;
      double p_divisions = ((*tft).width() - 42)/42;
```

```
double v_divisions = ((*tft).height() - 108)/14;

int p_max_graph = 0;
int p_min_graph = 10000;
int v_max_graph = 0;
int v_min_graph = 10000;
int v_division_value ;
float p_division_value ;

/// Clear LCD
(*tft).fillScreen(WHITE);
(*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

/// Print header
(*tft).setCursor(10, 10);
(*tft).setTextColor(BLACK);
(*tft).setTextSize(3);
(*tft).print("Calib. Curve");


///Print volume axis
(*tft).setCursor(5, (*tft).height()/2);
(*tft).setTextSize(2);
(*tft).print("V\r\n");
(*tft).setTextSize(1);
(*tft).print("(cc)");


/// These nested for loops search through the data_array to find max and minimum values of pressure
for(j = 0; j < NUMBER_VOLUME_CALIB_READS; j++)
{
    for(i = 0; i < NUMBER_VOLUME_CALIB_POINTS; i++)
    {

        if(((*((*(data_array + j)) + i)).volume) > max_volume)
        {
            max_volume = ((*((*((data_array) + j)) + i)).volume);

        }

        if(((*((*(data_array + j)) + i)).pressure) > max_pressure)
        {
            max_pressure = ((*((*(data_array + j)) + i)).pressure);
        }

        if(((*((*(data_array + j)) + i)).volume) < min_volume)
        {
            min_volume = ((*((*((data_array) + j)) + i)).volume);
        }
```

```
            if(((*((*(data_array + j)) + i)).pressure) < min_pressure)
            {
                min_pressure = ((*((*(data_array + j)) + i)).pressure);
            }

        }
    }


    /// Integer value for the max value of pressure on pressure axis
    p_max_graph = (max_pressure/1) + 2;

    /// Integer value for max volume on volume axis
    v_max_graph = max_volume/v_divisions;
    v_max_graph = (v_max_graph + 2) * v_divisions;


    /// Integer value for min value of pressure on pressure axis
    p_min_graph = (min_pressure/1);

    /// Integer value for min value of volume on volume axis
    v_min_graph = min_volume/v_divisions;
    v_min_graph = (v_min_graph - 1) * v_divisions;


    //Debugging print statements; uncomment as needed
    /* Serial.println(max_pressure); */
    /* Serial.println(min_pressure); */
    /* Serial.println(p_divisions); */
    /* Serial.println(p_max_graph); */
    /* Serial.println(p_min_graph); */

    /// Calculate the pressure divisions for pressure axis
    p_division_value = (p_max_graph - p_min_graph)/p_divisions;

    /// Calculate the volume divisions for volume axis
    v_division_value = (v_max_graph - v_min_graph)/v_divisions;



    /// Set cursor to print volume axis
    (*tft).setCursor(0, 48);
    (*tft).setTextSize(1);

    /// Print volume axis
    for (i = 0; i < v_divisions; i++)
    {
        (*tft).print("    ");
        (*tft).print(v_max_graph - (v_division_value * i));
        (*tft).print("\r\n\r\n");
```

```
}

(*tft).print("      ");

/// Print pressure axis
for (i = 0; i < p_divisions; i++)
{
     (*tft).print("   ");
     (*tft).print(p_min_graph + (p_division_value * i));
}
(*tft).setTextSize(2);

(*tft).print("\r\n          P (\"H2O)");

(*tft).drawLine(50, 36, 50, 200, BLACK);
(*tft).drawLine(24, 186, 320, 186, BLACK);


uint16_t x_coord_1 = 0;
uint16_t y_coord_1 = 0;

uint16_t x_coord_2 = 0;
uint16_t y_coord_2 = 0;

uint16_t color = BLUE;

for(j = 0; j < NUMBER_VOLUME_CALIB_READS; j++)
{
     for(i = 0; i < NUMBER_VOLUME_CALIB_POINTS; i++)
     {


          x_coord_1 = (((data_array[j][i].pressure) - p_min_graph) * (270/(p_max_graph-p_min_graph))) + 60;

          y_coord_1 = ((v_max_graph - (data_array[j][i].volume)) * (150/(v_max_graph-v_min_graph))) + 40;

          (*tft).fillCircle(x_coord_1, y_coord_1, 2, color);




     }

     cycle_color(&color);

}
```

```
        for(j = 0; j < NUMBER_VOLUME_CALIB_POINTS; j++)
        {
            if(j)
            {


                x_coord_1 = ((uint16_t)((avg_data_array[j].pressure) - p_min_graph) * (270/(p_max_graph-
p_min_graph))) + 60;

                y_coord_1 = (v_max_graph - (uint16_t)((avg_data_array[j].volume) * (150/(v_max_graph-
v_min_graph)))) + 40;

                x_coord_2 = ((uint16_t)((avg_data_array[j - 1].pressure) - p_min_graph) * (270/(p_max_graph-
p_min_graph))) + 60;

                y_coord_2 = (v_max_graph - (uint16_t)(avg_data_array[j - 1].volume) * (150/(v_max_graph-
v_min_graph))) + 40;


                (*tft).fillCircle(x_coord_1, y_coord_1, 2, BLACK);
                (*tft).fillCircle(x_coord_2, y_coord_2, 2, BLACK);

                (*tft).drawLine(x_coord_1, y_coord_1, x_coord_2, y_coord_2, BLACK);

            }
        }


        (*tft).print("\r\n Tap screen to continue");

        Point p;

        do
        {

            pinMode(13, OUTPUT);
            digitalWrite(13, HIGH);
            p = (*ts).getPoint();
            digitalWrite(13, LOW);
            pinMode(XM, OUTPUT);
            pinMode(YP, OUTPUT);
        } while(p.z < MINPRESSURE);

        char accept_header [] = "Accept or reject";
        char accept_text [] = "Would you like to accept\r\n or reject the volume\r\n calibration?";
        byte menu_opt;

        menu_opt = draw_continue_cancel(3, 2, accept_header, sizeof(accept_header), accept_text,
sizeof(accept_text), ACCEPT_REJECT, tft, ts);
```

```
        if(menu_opt == ACCEPT)
        {

            settings_write_int_volume(avg_data_array);

            settings_read_int_volume(15);
        }
        else
        {

            char do_new_header [] = "New Calibration?";
            char do_new_text [] = "Would you like to run a\r\n new calibration?";

            menu_opt = draw_yes_no_menu(3, 2, do_new_header, sizeof(do_new_header), do_new_text,
sizeof(do_new_text), tft, ts);

            if(menu_opt == YES)
            {
                return RERUN;
            }

        }




    }

    if(abort_status)
    {

        /// Clear LCD
        (*tft).fillScreen(WHITE);
        (*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

        /// Print header
        (*tft).setCursor(10, 10);
        (*tft).setTextColor(RED);
        (*tft).setTextSize(3);
        (*tft).print("TEST ABORTED");

        (*tft).print("\r\n\r\n");

        (*tft).setTextColor(BLACK);
        (*tft).setTextSize(2);

        (*tft).print(" Results not saved\r\n Tap screen to continue");
```

```
        Point p;

        do
        {

            pinMode(13, OUTPUT);
            digitalWrite(13, HIGH);
            p = (*ts).getPoint();
            digitalWrite(13, LOW);
            pinMode(XM, OUTPUT);
            pinMode(YP, OUTPUT);
        } while(p.z < MINPRESSURE);


    }




    if(leak)
    {

        /// Clear LCD
        (*tft).fillScreen(WHITE);
        (*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

        /// Print header
        (*tft).setCursor(10, 10);
        (*tft).setTextColor(RED);
        (*tft).setTextSize(3);

        (*tft).print("LEAK DETECTED");

        (*tft).print("\r\n\r\n");

        (*tft).setTextColor(BLACK);
        (*tft).setTextSize(2);

        (*tft).print(" Results not saved\r\n Tap screen to continue");



        Point p;

        do
        {

            pinMode(13, OUTPUT);
            digitalWrite(13, HIGH);
            p = (*ts).getPoint();
```

```
            digitalWrite(13, LOW);
            pinMode(XM, OUTPUT);
            pinMode(YP, OUTPUT);
        } while(p.z < MINPRESSURE);




    }




    return (0);

}
```

```
/**
 *
 * This function performs a leak check on the container in question. The function
 * first check to see whether the volume is up to a given sample pressure. If
 * it's not, then the function runs FPC101 to bring the sample volume to the
 * correct pressure. The function then shuts system valves and monitors the
 * container pressure for a period specified by LEAK_CHECK_WAIT_TIME. It
 * calculates the total mass flow into the container using compute_mass_flow_in,
 * and determines whether the mass flow rate through the seal is above the
 * LEAK_FREE_FLOW_RATE. If it is, then the leak_status byte is set high,
 * otherwise leak_status is set low. Note that this function IS NOT responsible
 * for opening valves after leak check to equilize pressure; calling function
```

```
 * must do this. (This might be changed in later versions.)
 *
 * @param double * target_pressure: Pointer to double that stores function target pressure
 * @param double * volume: Pointer to double which stores volume of container (cc)
 * @param double * mass_flow_in: Pointer to double which stores mass flow into container (scc)
 * @param double * mass_flow_out: Pointer to double which stores mass flow out of container (scc)
 * @param byte * abort_status: Pointer to abort_status variable in calling function
 * @param byte * leak_status: Pointer to leak_status variable in calling function
 * @param Touchscreen * ts: Pointer to touchscreen interface object
 * @param button * abort: Pointer to the abort button in the calling function. If there is
 * no abort button in the calling function, then a NULL pointer should be passed to this
 * function, otherwise a memory error will occur, or else the code won't compile.
 * @param byte test_case: Determines whether leak checking on test volume or on suction head
 *
 * @retval int 0: returns 0 upon successful completion
 */

int leak_check_ts(double * target_pressure, double * volume, double * mass_flow_in, double * mass_flow_out,  byte *
abort_status, byte * leak_status, Adafruit_TFTLCD * tft, button * abort, byte test_case, double * q_leak)
{

    /// PSUEDO-CODE INCLUDED IN INLINE COMMENTS BELOW:


    unsigned long start_time = 0;
    unsigned long last_time = 0;
    unsigned long current_time = 0;

    int print_status = 0;

    double pressure_init = 0.0;
    double pressure_fin = 0.0;
    double delta_P = 0.0;
    double rho = 0.0;
    double d_m = 0.0;
    double d_m_max = 0.0;
    double qrt_nom = 0.0;
    double vol_nom = 0.0;
    double del_P = 0.0;
    double q_leak_nomV = 0.0;

    double percent_diff = 0.0;
    double p_atm = 0.0;
    double h2o = 0.0;

    int CI_link;
    int PI_link;
    int FPC_link;
    int relief_valve;
    int capture_valve;
```

```
    /// Depending on what volume is being leak checked, the correct pressure transducer links are set.
    if(test_case == TEST_VOLUME)
    {
     CI_link = CI_101;
     PI_link = PI_101;
     FPC_link = CC_103;
     relief_valve = CC_101;
     capture_valve = CC_102;

    }
    else
    {
     CI_link = CI_102;
     PI_link = PI_102;
     FPC_link = CC_104;
     relief_valve = CC_106;
     capture_valve = CC_105;

    }

  /// Clear LCD
        (*tft).fillScreen(WHITE);
        (*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

/// Print header
        (*tft).setCursor(10, 10);
        (*tft).setTextColor(BLACK);
        (*tft).setTextSize(3);

        (*tft).print("Leak Check");
    Serial.println("Entered leak check function");

    // YOU SHOULD ADD THE CURRENT LEAK RATE CRITERIA
HERE**************************************************************************
        (*tft).setCursor(10, 40);
        (*tft).setTextSize(2);
        (*tft).print("\r\n Leak Rate Criteria: \r\n ");
        (*tft).print( LEAK_FREE_FLOW_RATE);
        (*tft).print(" cc/sec\r\n");

    delay(3000);

    /// Check status of abort button. If abort button is pressed, then *abort_status is
    /// set to zero. (This convention is repeated throughout function.)
    *abort_status = (*abort).get_status();


    /// Read atmospheric pressure
```

```
p_atm = read_pressure(CI_103, PI_103, 3);


/// Read current internal pressure
pressure_init = read_pressure(CI_link, PI_link, 3);




/// Calculate internal air density ratio (compared to atmospheric pressure
rho = pressure_init/p_atm;


/// Log start time
//start_time = micros();

//current_time = start_time;

//last_time = start_time;

//delay(1);



// DEBUG PRINT STATMENTS
Serial.print("P init\t");
Serial.println(pressure_init);

Serial.print("P atm\t");
Serial.println(p_atm);


if(test_case != TEST_VOLUME)
{


    //(*tft).setCursor(0, 120);
    //(*tft).setTextColor(BLACK);
    //(*tft).setTextSize(2);\
    //(*tft).print("q_leak_nomV:      scc/sec");



    //(*tft).setCursor(0, 150);
    //(*tft).setTextColor(BLACK);
    //(*tft).setTextSize(2);
    //(*tft).print("q_leak_nomV:      scc/sec");
```

```
}



/// Enter "while" loop. This loop runs for however long is specified in the \ref LEAK_CHECK_WAIT_TIME
/// variable
//while((current_time - start_time < LEAK_CHECK_WAIT_TIME) && !(*abort_status))
//{

    Serial.println("Running leak check...");


    //nominal value of canister
    h2o=500*0;

    //volume is the nominal value converted to cc, plus the system value 150cc
    vol_nom = (5230-h2o)+ 150.0;



    pressure_init = read_pressure(CI_link, PI_link, 3);
    Serial.print("Initial Pressure");
    Serial.println(pressure_init);


    Serial.println("Wait 10 seconds");
    delay(10000);

    pressure_fin = read_pressure(CI_link, PI_link, 3);
    Serial.print("Final Pressure");
    Serial.println(pressure_fin);

    /// Read atmospheric pressure
    p_atm = read_pressure(CI_103, PI_103, 3);
    Serial.print("Ambient Pressure");
    Serial.println(p_atm);


    /// Calculate drop in pressure from initial pressure to current pressure
    del_P = pressure_fin - pressure_init;
     Serial.print("Decay in Pressure");
     Serial.println(del_P);

     Serial.print("Calculated Volume");
     Serial.println(*volume);

    //q_leak_nomV = -100*del_P * vol_nom / p_atm / 10.0;
     //Serial.print("Leak Rate Nominal");
```

```
 //Serial.println(q_leak_nomV);


 *q_leak = -1*del_P * *volume / p_atm / 10.0;
 Serial.print("Leak Rate Calculated");
 Serial.println(*q_leak);

 //percent_diff = 100 * abs(q_leak_nomV-q_leak_calcV)/q_leak_nomV;
 //Serial.print("Percent Difference");
 //Serial.println(percent_diff);

/// Log current time
//current_time = micros();

// Compute total mass flow into container based on total pressure drop
//*mass_flow_in = compute_mass_flow_in(pressure_init, delta_P, *volume, rho);

/// Compute current average mass flow into container
//d_m = derivative(*mass_flow_in, 0, (current_time - start_time)/1000000.0);

/// Update current time
//last_time = current_time;

/// Update max mass flow rate
//if((d_m) < (d_m_max))
//{
      //d_m_max = (d_m);
//}



/// Print volume information to LCD if *tft is valid
if((tft != NULL) && (print_status != (millis()/SCREEN_PRINT_INTERVAL)))
{
  /// Clear LCD
//(*tft).fillScreen(WHITE);
//(*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

/// Print header
//(*tft).setCursor(10, 10);
//(*tft).setTextColor(BLACK);
//(*tft).setTextSize(3);

//(*tft).print("LEAK DETECTED");

//(*tft).setCursor(5, 120);
//(*tft).setTextColor(BLACK);
//(*tft).setTextSize(2);\
//(*tft).print(" leak_nom:");
```

```
//(*tft).setCursor(5, 150);
//(*tft).setTextColor(BLACK);
//(*tft).setTextSize(2);
//(*tft).print("leak_calc:");


//(*tft).setCursor(5, 180);
//(*tft).setTextColor(BLACK);
//(*tft).setTextSize(2);
//(*tft).print("  % diff:       ");



      //(*tft).setCursor(200, 120);
      //(*tft).setTextColor(BLACK);
      //(*tft).setTextSize(2);
      //(*tft).print(q_leak_nomV);

      //(*tft).setCursor(200, 150);
      //(*tft).setTextColor(BLACK);
      //(*tft).setTextSize(2);
      //(*tft).print(q_leak);

      //(*tft).fillRect(110, 90, 90, 80, WHITE);
      //(*tft).setTextColor(BLACK);
      //(*tft).setTextSize(2);
      //(*tft).setCursor(200, 180);
      //(*tft).print(percent_diff);


      delay(100);
      //Serial.print(read_pressure(CI_link, PI_link, 3));
      //(*tft).setCursor(110, 120);
      //if((abs(delta_P) > 10) || (delta_P < 0.0))
      //{
            //print_scientific(tft, d_m, 1);
      //}
      //else
      //{
            //print_scientific(tft, d_m);
      //}
      //(*tft).setCursor(110, 150);
      //(*tft).print((*mass_flow_out) + (*mass_flow_in));

}
```

```
            //print_status = millis()/SCREEN_PRINT_INTERVAL;

            //*abort_status = (*abort).get_status();

            //delay(1);
//
      //}


      //Serial.println(d_m_max);
      //Serial.print(LEAK_FREE_FLOW_RATE);

      if ((*q_leak) > (LEAK_FREE_FLOW_RATE))
      {

            *leak_status = LEAKY;
            return LEAKY;

      }

      *leak_status = SECURE;


      return SECURE;

}
```

```
//This function secures the test apparatus to the container.



int secure_hardware_ts(byte * abort_status,Adafruit_TFTLCD * tft, button * abort, double * final_pressure)
{
```

```
/// Check status of abort button. If abort button is pressed, then *abort_status is
/// set to zero. (This convention is repeated throughout function.)
*abort_status = (*abort).get_status();



Serial.println("Entered secure hardware function");
int print_status = 0;
int max_p = settings_read_pressure1();

unsigned long start_time = 0;

float suction_pressure = 0.0;
float sample_pressure = 0.0;

Serial.print("Target pressure:\t");
Serial.println(max_p);

if(!(*abort_status))
{
    Serial.println("Turning on pump and valve");

    /// Release any residual pressure on sample chamber.
    digitalWrite(CC_101, ON);

    delay(RELAY_LAG_TIME);

    digitalWrite(CC_102, ON);


    delay(1000);



    digitalWrite(CC_101, OFF);
    delay(RELAY_LAG_TIME);
    digitalWrite(CC_102, OFF);

    delay(RELAY_LAG_TIME);

    //Open VC102, turn on FPC102
    digitalWrite(CC_105, ON);
    delay(500);
    digitalWrite(CC_104, ON);

}

delay(500);

Serial.println("Reading pressure...");
```

```
    Serial.println("calculting pressure");

    suction_pressure = read_pressure(CI_102, PI_102);
    sample_pressure = read_pressure(CI_101, PI_101);

    Serial.println("Entering while loop");

    start_time = millis();

    while (((millis() - start_time) < 5000)&&(!(*abort_status)))
    {

        suction_pressure = read_pressure(CI_102, PI_102);
        sample_pressure = read_pressure(CI_101, PI_101);


        Serial.println("iteration");
        Serial.print("Suction pressure:\t");
        Serial.println(suction_pressure);

        Serial.print("Sample pressure:\t");
        Serial.println(sample_pressure);

        *final_pressure = suction_pressure;

        /// Check on abort button, update *abort_status
        if((*abort).get_status())
        {
            *abort_status = USER_ABORT;
        }


        /// If the function has been passed a valid LCD object, function will update volume read status to screen
        /// Also, the frequency of updates is determined by SCREEN_PRINT_INTERVAL
        if ((tft != NULL) && (print_status != (millis()/SCREEN_PRINT_INTERVAL)))
        {
            (*tft).fillRect(110, 90, 90, 120, WHITE);

            (*tft).setTextColor(BLACK);
            (*tft).setTextSize(2);
            (*tft).setCursor(130, 90);
            (*tft).print(sample_pressure);


            (*tft).setTextColor(BLACK);
            (*tft).setTextSize(2);
            (*tft).setCursor(130, 120);
            (*tft).print(suction_pressure);
```

```
        }

        /// Update the print_status variable to prevent function from printing too often
        print_status = millis()/SCREEN_PRINT_INTERVAL;


        /// Check to see whether the sample pressure is rising. If it is, then the O-ring is insecure and the suction sealing process should be aborted.
        if (sample_pressure > 10.0)
        {

            *abort_status = ORING_FAILED;
            Serial.println("O-ring failure");

            digitalWrite(CC_101, OFF);
            digitalWrite(CC_102, OFF);
            digitalWrite(CC_103, OFF);
            digitalWrite(CC_104, OFF);
            digitalWrite(CC_105, OFF);
            digitalWrite(CC_106, OFF);
        }


    }


    /// If loop exits with a positive abort, print that volume read is aborting
    if((*abort_status))
    {
        (*tft).setCursor(0, 210);
        (*tft).print("Aborting...");
    }




    /// Return 0 upon successful completion
    return 0;

}
```

```
/**
 *
 *  This function measures the volume of a container. Function manages all sensor and
 *  hardware interfacing. Note that the measure_volume_ts is a variation on the
 *  original measure_volume function (legacy, no longer included in source code), with
 *  touchscreen and LCD interfacing capabilities included. This is a general volume
 *  measurement function and is tied both to standard filter measurements as well as
 *  to internal system volume calibration.
 *
 *  @param double target_pressure: Differential vacuum pressure to which the system will
 *  attempt to draw the sample volume ("H2O)
 *  @param int run_time: Maximum amount of time for which the system will attempt to bring
 *  the sample volume's pressure down to the target pressure (seconds)
 *  @param double v_int: Internal volume of system plumbing. Note, if doing an internal
 *  volume calibration, then calling function should pass zero for this variable
 *  @param double * volume: Pointer to the volume variable in the calling function (scc)
 *  @param byte * leak: Pointer to the leak variable in calling function. If a leak is
 *  detected during the course of the volume measurement, then this variable is set to 1.
 *  Otherwise it is held at 0.
 *  @param byte * abort_status: Pointer to the abort_status variable in calling function. If
 *  volume measurement is aborted mid-measurement, this variable is set to 1, otherwise it
 *  is held at 0.
 *  @param byte do_leak_check: Tells code whether to do a leak check.
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *  @param button * abort: Pointer to the abort button in the calling function. If there is
 *  no abort button in the calling function, then a NULL pointer should be passed to this
 *  function, otherwise a memory error will occur, or else the code won't compile.
 *
 *  @retval int 0: returns 0 upon successful completion
 *
 *  @todo Add real leak-checking criteria (Kirk Reeves, OCFT logbook 1, p. 56). Currently
 *  the leak-check criteria are completely arbitrary, and shouldn't be used for determining
 *  O-ring status.
 *
 */

int measure_volume_ts(double target_pressure, int run_time, double v_int, double * volume, byte * leak, byte *
abort_status, byte do_leak_check,  Adafruit_TFTLCD * tft, button * abort, double * q_leak)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    unsigned long start_time = 0;
    unsigned long last_time = 0;
    unsigned long stop_time;
```

```
int print_status = 0;


//FLOAT TYPE VARIABLES

double air_density = 0.0;
double atm_pressure = 0.0;
double delta_mass_in = 0.0;
double delta_mass_out = 0.0;
double mass_flow = 0.0;
double mass_flow_last = 0.0;
double pressure_in = 0.0;
double pressure_initial = 0;
double pressure_out = 0.0;


Serial.println("Entered volume function");

/// Set leak variable to zero
*leak = 0;

/// Check status of abort button. If abort button is pressed, then *abort_status is
/// set to zero. (This convention is repeated throughout function.)
*abort_status = (*abort).get_status();


/// Open VC101, VC102 allow system pressure to equalize, then close valves
digitalWrite(CC_101, ON);
delay(RELAY_LAG_TIME);
digitalWrite(CC_102, ON);

Serial.println("equalizing pressure");

/// If abort button is pressed during pressure equalization, then while loop exits before pressure has equalized.
while((read_pressure(CI_101, PI_101) > 0.5) && !(*abort_status))
{
    *abort_status = (*abort).get_status();
    Serial.println("Checking abort status");
}

Serial.print("Exited while loop");
delay(RELAY_LAG_TIME);

/// Close VC101, VC102
digitalWrite(CC_101, OFF);
delay(RELAY_LAG_TIME);
digitalWrite(CC_102, OFF);
delay(RELAY_LAG_TIME);
```

```
    Serial.println("Turning on valve and pump");

    /// If read is not aborted, enter main volume measurement sequence
    if(!(*abort_status))
    {

        //Open VC102, turn on FPC101
        digitalWrite(CC_102, ON);
        delay(RELAY_LAG_TIME);
        digitalWrite(CC_103, ON);

    }




    /// Log time at which pump was activated, log initial pressure
    last_time = micros();
    start_time = last_time;

    pressure_initial = read_pressure(CI_101, PI_101);


    /// Enter loop that runs until pressure desired target pressure
    while((pressure_out - pressure_initial < target_pressure) && (!(*abort_status)))
    {

        Serial.println("Entered while loop");

        /// Read pressure and mass flow (these variables will need to be renamed, as we are adding more pressure
sensors
        pressure_out = read_pressure(CI_101, PI_101);

        mass_flow = read_mass_flow(CI_104, FI_101);

        if(mass_flow < 0.3)
        {
            mass_flow = 0;
        }

        /// Calculate mass flow out, using trapezoidal integration
        delta_mass_out = delta_mass_out + trap_integrate(mass_flow, mass_flow_last, (micros()-
last_time)/(1000000.0));

        mass_flow_last = mass_flow;
```

```
        last_time = micros();

        /// Check to see how long the volume read has been going. If it has been going longer than the max run time,
exit while loop
        if ((last_time - start_time)/1000000 > run_time)
        {
            *leak = 1;

            break;
        }


        /// Check on abort button, update *abort_status
        *abort_status = (*abort).get_status();



        /// If the function has been passed a valid LCD object, function will update volume read status to screen
        /// Also, the frequency of updates is determined by SCREEN_PRINT_INTERVAL
        if ((tft != NULL) && (print_status != (millis()/SCREEN_PRINT_INTERVAL)))
        {
            (*tft).fillRect(110, 90, 90, 120, WHITE);

            (*tft).setTextColor(BLACK);
            (*tft).setTextSize(2);
            (*tft).setCursor(110, 90);
            (*tft).print(pressure_out);
            (*tft).setCursor(110, 120);
            (*tft).print(mass_flow);
            (*tft).setCursor(110, 150);
            (*tft).print(delta_mass_out);
        }

        /// Update the print_status variable to prevent function from printing too often
        print_status = millis()/SCREEN_PRINT_INTERVAL;



        //Print out data for trouble-shooting purposes and data collection... uncomment as needed
        //    Serial.print("Timestamp =\t");
        //    Serial.print(millis());
        //    Serial.print("\tPressure =\t");
        //    Serial.print(pressure_out);
        //    Serial.print("\tMass flow rate =\t");
        //    Serial.print(mass_flow);
        //    Serial.print("\tTotal mass out =\t");
        //    Serial.println(delta_mass_out);
```

```
    }


    /// If loop exits with a positive abort, print that volume read is aborting
    if((*abort_status))
    {
        (*tft).setCursor(0, 210);
        (*tft).print("Aborting...");
    }

    /// Close solenoid valve, wait 0.1 seconds, then turn off the pump.
    digitalWrite(CC_102, OFF);

    /// Record the time at which the valve was close
    stop_time = micros();

    delay(RELAY_LAG_TIME);
    digitalWrite(CC_103, OFF);

    /// Reset value of mass_flow_last variable
    mass_flow_last = 0.0;

    /// Read current atmospheric pressure;
    atm_pressure = read_pressure(CI_103, PI_103);


    /// Calculate normalized air density based only on atmospheric pressure. Note that this
    /// assumes STP conditions for temperature and humidity and only takes atmospheric pressure
    /// into account. We may need to add a temperature and humidity sensor down the road, depending
    /// on final applications of product.
    air_density = atm_pressure/406.64355;


    /// If function has a valid LCD pointer and read has not been aborted, function prints final volume results to
touchscreen
    if((tft != NULL) && (!(*abort_status)))
    {

        /// Compute volume
        *volume = compute_volume(atm_pressure, pressure_out - pressure_initial, delta_mass_out, air_density) -
v_int;


        /// Print volume information to LCD
        (*tft).fillRect(110, 90, 90, 120, WHITE);
        (*tft).setTextColor(BLACK);
        (*tft).setTextSize(2);
```

```
        (*tft).setCursor(110, 90);
        (*tft).print(pressure_out);
        (*tft).setCursor(110, 120);
        (*tft).print(mass_flow);
        (*tft).setCursor(110, 150);
        (*tft).print(delta_mass_out);
        (*tft).setCursor(110, 180);
        (*tft).print(*volume);

    }



    /// If leak has not already been detected and read was not aborted, then proceed to check integrity of
    /// O-ring. Note that the whole "leak-checking" parameters are so far completely arbitrary. Need to
    /// implement the helium leak criteria from OCFT logbook 1, page 56
    if(!(*leak) && (!(*abort_status)) && (do_leak_check == YES))
    {
        /// Compute air density again (this is redundant)
        atm_pressure = read_pressure(CI_103, PI_103);
        air_density = atm_pressure/406.64355;

        /// Compute volume again, passing it to *volume
        *volume = compute_volume(atm_pressure, pressure_out - pressure_initial, delta_mass_out, air_density) -
v_int;


        leak_check_ts(&target_pressure, volume, &delta_mass_in, &delta_mass_out, abort_status, leak, tft, abort,
TEST_VOLUME, q_leak);

    }

    if(do_leak_check == NO)
    {

        delay(RELAY_LAG_TIME);

    }

    digitalWrite(CC_104, OFF);


    /// Open VC101, VC102 allow system pressure to equalize
    digitalWrite(CC_101, ON);
    delay(RELAY_LAG_TIME);
    digitalWrite(CC_102, ON);
    delay(RELAY_LAG_TIME);
    digitalWrite(CC_106, ON);
    delay(RELAY_LAG_TIME);
    digitalWrite(CC_105, ON);
```

```
    /// Keep valves open until system pressure has equalized, or until user aborts
    while((read_pressure(CI_102, PI_102) > 0.5) && !(*abort_status))
    {
        *abort_status = (*abort).get_status();
    }

    delay(RELAY_LAG_TIME);

    /// Close VC101, VC102
    digitalWrite(CC_101, OFF);
    delay(RELAY_LAG_TIME);
    digitalWrite(CC_102, OFF);
    delay(RELAY_LAG_TIME);
    digitalWrite(CC_105, OFF);
    delay(RELAY_LAG_TIME);
    digitalWrite(CC_106, OFF);



    /// Return 0 upon successful completion
    return (0);

}
```

```
/**
 *
 *  This function runs is responsible for running a typical filter test. Note that
 *  this function is a variation on the original "run_test" function (legacy, no
 *  longer included in source code), with touchscreen support added. The function
 *  first draws test information on the touchscreen and creates an abort button. It
 *  then passes control to the measure_volume_ts function. Upon completion of
 *  measure_volume_ts, the function displays data from the test to user on LCD,
 *  including measured volume, and determined filter and O-ring status (if available).
 *  Function prompts user to continue through a continue button.
 *
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int 0: returns 0 upon successful completion
 *
 */


int run_test_ts(Adafruit_TFTLCD * tft, TouchScreen * ts)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-


    /// Create test variables
    double system_volume = 0;
    byte leak = 0;
    byte clogged = 0;
    byte abort_status = 0;



    char abort_button_text [] = "Abort";
    char done_button_text [] = "Done";

    byte i = 0;

    double suction_pressure = 0;



    /// Create and draw abort button
```

```
button abort_button((*tft).width() - 90, (*tft).height() - 40, abort_button_text, sizeof(abort_button_text), 2, tft, ts);



/// Display secure hardware header to LCD
(*tft).fillScreen(WHITE);

(*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

(*tft).setCursor(10, 10);
(*tft).setTextColor(BLACK);
(*tft).setTextSize(3);
(*tft).print("Filter Test");

(*tft).setCursor(0, 50);
(*tft).setTextColor(BLACK);
(*tft).setTextSize(2);
(*tft).print(" Securing hardware to lid");


(*tft).setCursor(0, 90);
(*tft).setTextColor(BLACK);
(*tft).setTextSize(2);
(*tft).print("  Sample P:       \"H2O");



(*tft).setCursor(0, 120);
(*tft).setTextColor(BLACK);
(*tft).setTextSize(2);\
(*tft).print(" Suction P:       \"H2O");

abort_button.draw(WHITE, BLACK, RED);


suction_pressure = settings_read_pressure1();

secure_hardware_ts(&abort_status, tft, &abort_button, &suction_pressure);



/// Display test information header to LCD
(*tft).fillScreen(WHITE);

(*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

(*tft).setCursor(10, 10);
(*tft).setTextColor(BLACK);
(*tft).setTextSize(3);
(*tft).print("Filter Test");
```

```
    (*tft).setCursor(0, 50);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print(" Test in progress...");


    (*tft).setCursor(0, 90);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print(" Delta P:       \"H2O");



    (*tft).setCursor(0, 120);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);\
    (*tft).print("      dm:       scc/s");



    (*tft).setCursor(0, 150);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print(" Delta m:       scc");

    (*tft).setCursor(0, 180);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print("  Volume:       cc");


    abort_button.draw(WHITE, BLACK, RED);


    /// Function passes control to measure_volume_ts()
double q_leak = 0;
    if(!abort_status)
    {

        abort_status = abort_button.get_status();


        measure_volume_ts(settings_read_pressure(), settings_read_time(),
settings_read_int_volume(settings_read_pressure()), &system_volume, &leak, &abort_status, YES, tft, &abort_button,
&q_leak);

    }
```

```
// Debugging print statements; comment or uncomment as needed
Serial.println(settings_read_int_volume(settings_read_pressure()));
Serial.println(system_volume);


/// Clear display
(*tft).fillScreen(WHITE);


/// Create and draw "Done" button
button done_button((*tft).width() - 90, (*tft).height() - 40, done_button_text, sizeof(done_button_text), 2, tft, ts);

done_button.draw(WHITE, BLACK, BLACK);


/// Display test data
(*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

(*tft).setCursor(10, 10);
(*tft).setTextColor(BLACK);
(*tft).setTextSize(3);
(*tft).print("Filter Test");


(*tft).setTextSize(2);

(*tft).setCursor(0, 50);

(*tft).println(" Test results: \r\n");

digitalWrite(CC_106, OFF);

delay(RELAY_LAG_TIME);

digitalWrite(CC_105, OFF);

delay(RELAY_LAG_TIME);

digitalWrite(CC_104, OFF);

delay(RELAY_LAG_TIME);

digitalWrite(CC_103, OFF);

delay(RELAY_LAG_TIME);

digitalWrite(CC_102, OFF);

delay(RELAY_LAG_TIME);
```

```
    digitalWrite(CC_101, OFF);


    /// If run was not aborted, display data.
    if(!abort_status)
    {

        /// Display measured volume
        (*tft).print(" Volume: ");
        (*tft).print(system_volume);
        (*tft).print(" cc\r\n");

        (*tft).print("\r\n Leak Rate: ");
```

```
        (*tft).print(q_leak);
        (*tft).print(" cc/sec\r\n");

        (*tft).print("\r\n Filter status: ");


        /// If filter is clogged, display "CLOGGED" message, otherwise display "GOOD" message
        if(system_volume < CLOGGED_FILTER_THRESHOLD)
        {
            (*tft).setTextColor(RED);
            (*tft).println("CLOGGED");
            clogged = 1;
        }
        else
        {
            (*tft).setTextColor(GREEN);
            (*tft).println("GOOD");
        }




        /// Display O-ring status
        (*tft).setTextColor(BLACK);
        (*tft).print("\r\n O-ring status: ");

        /// If the filter was clogged, O-ring status is "UNKNOWN"
        if(clogged)
        {
            (*tft).setTextColor(YELLOW);
            (*tft).print("UNKNOWN\r\n");
        }

        /// Otherwise, determine O-ring status based on leakiness of container, display results accordingly
        else
        {
            if(leak)
            {
                (*tft).setTextColor(RED);
                (*tft).print("LEAKY\r\n");
            }
            else
            {
                (*tft).setTextColor(GREEN);
                (*tft).print("SECURE\r\n");
            }
        }

    }
```

```
    /// If run was aborted, display simple abort message.
    else
    {

        if(abort_status == USER_ABORT)
        {

            (*tft).fillScreen(WHITE);
            (*tft).setTextColor(RED);
            (*tft).print(" Test aborted by user\r\n\r\n");
            (*tft).print(" Results invalid.");
        }


        if(abort_status == ORING_FAILED)
        {

            (*tft).fillScreen(WHITE);
            (*tft).setTextColor(RED);
            (*tft).print(" System O-Ring Failed!\r\n Recheck seals before proceeding.\r\n");
            (*tft).print(" Results invalid.");
        }


    }
    done_button.draw(WHITE, BLACK, BLACK);

    /// Do nothing until user presses the "Done" button
    while(!done_button.get_status())
    {
        ;
    }



    /// Return zero upon successful completion
    return (0);


}




//@}
```

```
/**
 *
 *  This function runs is responsible for SECURING THE TEST HARDWARE. Note that
 *  this function is a variation on the original "run_test" function (legacy, no
 *  longer included in source code), with touchscreen support added. The function
 *  first draws test information on the touchscreen and creates an abort button. It
 *  then passes control to the measure_volume_ts function. Upon completion of
 *  measure_volume_ts, the function displays data from the test to user on LCD,
 *  including measured volume, and determined filter and O-ring status (if available).
 *  Function prompts user to continue through a continue button.
 *
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int 0: returns 0 upon successful completion
 *
 *///double target_pressure, int run_time, double v_int, double * volume, byte * leak, byte * abort_status, byte
do_leak_check,  Adafruit_TFTLCD * tft, button * abort)


int run_secure_hardware_ts(Adafruit_TFTLCD * tft, TouchScreen * ts)
{




  byte abort_status = 0;
    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-


      char abort_button_text [] = "Abort";
      char done_button_text [] = "Done";

      byte i = 0;

      /// Display test information header to LCD
      (*tft).fillScreen(WHITE);

      (*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

      (*tft).setCursor(10, 10);
      (*tft).setTextColor(BLACK);
      (*tft).setTextSize(3);
      (*tft).print("Secure Test");
```

```
    (*tft).setCursor(0, 50);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print(" Test in progress...");


    (*tft).setCursor(0, 90);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(2);
    (*tft).print(" Delta P:       \"H2O");




    /// Create and draw abort button
    button abort_button((*tft).width() - 90, (*tft).height() - 40, abort_button_text, sizeof(abort_button_text), 2, tft, ts);

    abort_button.draw(WHITE, BLACK, RED);


    //settings_read_pressure(), settings_read_time(), settings_read_int_volume(settings_read_pressure()),
&system_volume, &leak, &abort_status, YES, tft, &abort_button);



    ///Passes control to the secure hardware function.....
    double secure_pressure = 0;
    double delta_mass_in = 0;
    double delta_mass_out = 0;
    byte leak;
    double suction_head_volume = SUCTION_HEAD_VOLUME;
    secure_hardware_ts(&abort_status, tft, &abort_button, &secure_pressure);

   double q_leak=0;

    if(!abort_status)
    {
        leak_check_ts(&secure_pressure, &suction_head_volume, &delta_mass_in, &delta_mass_out, &abort_status,
&leak, tft, &abort_button, SUCTION_VOLUME, &q_leak);
    }
    /// Clear display
    (*tft).fillScreen(WHITE);


    /// Create and draw "Done" button
    button done_button((*tft).width() - 90, (*tft).height() - 40, done_button_text, sizeof(done_button_text), 2, tft, ts);

    done_button.draw(WHITE, BLACK, BLACK);
```

```
/// Display test data
(*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

(*tft).setCursor(10, 10);
(*tft).setTextColor(BLACK);
(*tft).setTextSize(3);
(*tft).print("Secure Test");


(*tft).setTextSize(2);

(*tft).setCursor(0, 50);




/// If run was not aborted, display data.
if(!abort_status)
{

      /// Display measured volume
      (*tft).print(" Secured to: ");
      (*tft).print(secure_pressure);
      (*tft).print( "\"H2O\r\n");

}



/// If run was aborted, display simple abort message.
else
{

      (*tft).fillScreen(WHITE);
      (*tft).setTextColor(RED);
      (*tft).print(" Secure aborted...\r\n\r\n");
      (*tft).print(" Test is not secure");

}
done_button.draw(WHITE, BLACK, BLACK);

/// Do nothing until user presses the "Done" button
while(!done_button.get_status())
{
      ;
}
```

```
    /// Return zero upon successful completion
    return (0);


}
```

/// This function is resposable for RELEASING TEST HARDWARE.

```
int run_release_hardware_ts(Adafruit_TFTLCD * tft, TouchScreen * ts)
{



    char done_button_text [] = "Done";

    byte i = 0;


    /// Display test information header to LCD
    (*tft).fillScreen(WHITE);

    (*tft).fillRect(0, (36), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(3);
    (*tft).print("Release Hardware");

    (*tft).setCursor(0, 50);
    (*tft).setTextColor(BLACK);
```

```
    (*tft).setTextSize(2);
    (*tft).print(" In progress...");

 //Procedure for releasing the hardware
        digitalWrite(CC_106, ON);
        delay(RELAY_LAG_TIME);
        digitalWrite(CC_105, ON);

        delay(1500);

        digitalWrite(CC_106, OFF);
        delay(RELAY_LAG_TIME);
        digitalWrite(CC_105, OFF);



  /// Display hardware has been released
        (*tft).print(" Hardware Released ");

  /// Create and draw "Done" button
    button done_button((*tft).width() - 90, (*tft).height() - 40, done_button_text, sizeof(done_button_text), 2, tft, ts);

    done_button.draw(WHITE, BLACK, BLACK);



    /// Do nothing until user presses the "Done" button
    while(!done_button.get_status())
    {
        ;
    }


    /// Return zero upon successful completion
    return (0);

}
```

```
/************************************/
/*  Touchscreen Interface Functions  */
/************************************/
```

```
/**
 * @anchor ts_interface
 * @name Touchscreen Interface Functions
 *
 *  This category contains functions which are responsible solely for
 *  touchscreen interfacing, both displaying data to touchscreen and receiving
 *  user input from touchscreen. These functions often pass user input directly
 *  to the logic flow control functions of the software.
 *
 *
 */


//@{



/**
 *
 * This is a simple function designed to cycle a color variable through standard
 * color values. It cycles the color variable to the next pre-determined color
 * each time it is called.
 *
 * @param uint16_t * color Pointer to the current color value
 *
 * @retval int returns 0 upon completion
 *
 */

int cycle_color(uint16_t * color)
{

      switch (*color)
      {

      case BLUE:
            *color = RED;
            break;

      case RED:
            *color = GREEN;
            break;

      case GREEN:
            *color = CYAN;
```

```
            break;

        case CYAN:
            *color = MAGENTA;
            break;

        case MAGENTA:
            *color = YELLOW;
            break;

        case YELLOW:
            *color = BLUE;

        }

        return (0);
}
```

```
/**
 *
 *  This function is responsible for drawing numeric value input menus to the touchscreen,
 *  and reporting the entered value to the user. The function accepts a custom structure
 *  of data that contains all the necessary information to draw the touchscreen. The
 *  function allows the calling function to enter a current "starting" value, increment
```

```
 *  value, max and min values, warning values, custom units, and a custom header for the
 *  menu. The function draws the menu to the touchscreen, and waits for user input. Function
 *  blocks user from setting variable outside of the maximum and minimum values, and warns
 *  user if they are setting value outside of warning limits (with red text). When user is
 *  done setting variable, function returns value set by user to calling function.
 *
 *  @param scroll_menu_input menu_data: Contains all the data needed to display scroll menu
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval double value: returns the value entered by user to the calling function
 *
 *  @todo Add support for negative numbers, numbers larger than 1000. Specifically, the box
 *  surrounding the variable needs to be dynamically sized based on the size and sign of the
 *  number. Also, possibly could add support for calling function to determine how many
 *  numbers they want to display, how many decimal places, etc. This is only needed if the
 *  function is to be used for a more general case though.
 *
 */

double draw_scroll_menu (scroll_menu_input menu_data, Adafruit_TFTLCD * tft, TouchScreen * ts)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    char plus_text [] = "+";
    char minus_text [] = "-";
    char exit_text [] = " Done ";



    byte i = 0;
    byte places = 0;
    double value = menu_data.val_init;


    /// Clear screen
    (*tft).fillScreen(WHITE);

    /// Draw menu
    (*tft).fillRect(0, ((menu_data.header_size) * 12), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(menu_data.header_size);


    /// Print user-inputted menu title
    for(i = 0; i < menu_data.scroll_size; i++)
    {
```

```
        (*tft).write(*((menu_data.scroll_title) + i));

    }

    (*tft).setCursor(10, ((menu_data.header_size) * 12) + ((menu_data.button_size) * 20) + 50);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(menu_data.button_size - 1);


    /// Print user-inputted units
    for(i = 0; i < menu_data.unit_size; i++)
    {

        (*tft).write(*((menu_data.units) + i));

    }


    ///Draw menu rectangles
    (*tft).fillRoundRect(10, ((menu_data.header_size) * 12) + 50, ((menu_data.button_size) * 5 * 6),
((menu_data.button_size) * 7 * 2), 5, WHITE);
    (*tft).drawRoundRect(10, ((menu_data.header_size) * 12) + 50, ((menu_data.button_size) * 5 * 6),
((menu_data.button_size) * 7 * 2), 5, BLACK);
    (*tft).setCursor(10 + ((menu_data.button_size) * 5), ((menu_data.header_size) * 12) + ((menu_data.button_size) *
3) + 50);



    /// Print initial value in appropriate box

    /// If the value is above or below warning levels, print in red, otherwise print in black
    if(value > menu_data.warning_max)
    {
        (*tft).setTextColor(RED);
    }
    else if(value < menu_data.warning_min)
    {
        (*tft).setTextColor(RED);
    }
    else
    {
        (*tft).setTextColor(BLACK);
    }


    (*tft).setTextSize(menu_data.button_size);

    /// Function supports decimals and integers. Following if statements control the decimal place printed based on size
of number
```

```
    if((int)(value/100))
    {
        (*tft).print(value, 0);
    }
    else if ((int)(value/10))
    {
        (*tft).print(value, 1);
    }
    else
    {
        (*tft).print(value, 2);
    }


    /// Create "+", "-", and "Done" button objects
    button button_plus(((*tft).width() - (menu_data.button_size * 25)), (menu_data.header_size * 12) +
(menu_data.button_size * 3) + 10, plus_text, sizeof(plus_text), menu_data.button_size, tft, ts);

    button button_minus(((*tft).width() - (menu_data.button_size * 25)), (menu_data.header_size * 12) +
(menu_data.button_size * 23) + 10, minus_text, sizeof(minus_text), menu_data.button_size, tft, ts);

    button button_exit(((*tft).width() - (menu_data.button_size * 35)), (*tft).height() - (menu_data.button_size * 20),
exit_text, sizeof(exit_text), menu_data.button_size - 1, tft, ts);


    /// Draw buttons
    button_plus.draw(WHITE, BLACK, BLACK);
    button_minus.draw(WHITE, BLACK, BLACK);
    button_exit.draw(WHITE, BLACK, BLACK);



    /// Enter while loop in which function waits for user to finish entering number
    while(1)
    {


        /// This if statement executes if the "+" button is pressed
        if(button_plus.get_status())
        {

            /// Increment the value by one "increment"
            value = value + menu_data.increment;

            /// If the value is too high, set the value to max value
            if(value > menu_data.val_max)
            {
                value = menu_data.val_max;
            }
```

```
            /// Draw white rectangle over old value and set cursor to print new value
            (*tft).fillRoundRect(10, (menu_data.header_size * 12) + 50, (menu_data.button_size * 5 * 6),
(menu_data.button_size * 7 * 2), 5, WHITE);

            (*tft).drawRoundRect(10, (menu_data.header_size * 12) + 50, (menu_data.button_size * 5 * 6),
(menu_data.button_size * 7 * 2), 5, BLACK);

            (*tft).setCursor(10 + (menu_data.button_size * 5), (menu_data.header_size * 12) +
(menu_data.button_size * 3) + 50);


            /// IF the new value is above the warning max or below the warning min, set text color to red, otherwise
set to black
            if(value > menu_data.warning_max)
            {
                (*tft).setTextColor(RED);
            }
            else if(value < menu_data.warning_min)
            {
                (*tft).setTextColor(RED);
            }
            else
            {
                (*tft).setTextColor(BLACK);
            }

            /// Set text size
            (*tft).setTextSize(menu_data.button_size);

            /// Determine decimal place to print value to based on value of number
            if((int)(value/100))
            {
                (*tft).print(value, 0);
            }
            else if ((int)(value/10))
            {
                (*tft).print(value, 1);
            }
            else
            {
                (*tft).print(value, 2);
            }
        }


        /// This if statement executes if the "-" button is pressed
        if(button_minus.get_status())
        {

            /// Decrement the value by one "increment"
```

```
            value = value - menu_data.increment;

            /// If the value is too low, set the value to min value
            if(value < menu_data.val_min)
            {
                  value = menu_data.val_min;
            }

            /// Draw white rectangle over old value and set cursor to print new value
            (*tft).fillRoundRect(10, (menu_data.header_size * 12) + 50, (menu_data.button_size * 5 * 6),
(menu_data.button_size * 7 * 2), 5, WHITE);
            (*tft).drawRoundRect(10, (menu_data.header_size * 12) + 50, (menu_data.button_size * 5 * 6),
(menu_data.button_size * 7 * 2), 5, BLACK);
            (*tft).setCursor(10 + (menu_data.button_size * 5), (menu_data.header_size * 12) +
(menu_data.button_size * 3) + 50);


            /// If new value is above the warning max or below the warning min, set the text color to red, otherwise
set to black
            if(value > menu_data.warning_max)
            {
                  (*tft).setTextColor(RED);
            }
            else if(value < menu_data.warning_min)
            {
                  (*tft).setTextColor(RED);
            }
            else
            {
                  (*tft).setTextColor(BLACK);
            }

            /// Set text size
            (*tft).setTextSize(menu_data.button_size);

            /// Determine decimal place to which to print value, based on value of
            /// number (note, menu currently doesn't support values larger than 1000)
            if((int)(value/100))
            {
                  (*tft).print(value, 0);
            }
            else if ((int)(value/10))
            {
                  (*tft).print(value, 1);
            }
            else
            {
                  (*tft).print(value, 2);
            }
```

```
            }

            /// This if statement executes if user presses the "Done"/"Exit" button
            if(button_exit.get_status())
            {
                /// Function returns the user-entered value to calling function
                return(value);
            }









        }

        // This line of code will never run... EVER!!!
        return(0);

}
```

```
/**
 *
 *  This function draws a "Continue"/"Cancel" menu to the LCD. It then
 *  waits for user input.  Function allows for the text of the menu to
 *  be determined by the calling function. However, the function does not
 *  include support for rejecting text that is too large. So if the
 *  calling function sends text that is too big for the screen, the text
 *  will run over into the button area. Also, note that function includes
 *  no support for word-wrapping. Currently it is up to the calling
 *  function to ensure that the text has newline characters in place where
 *  needed for proper word wrapping. (Menu will TEXT wrap, but not word
 *  wrap.)
 *
 *  @param uint16_t header_size: Size in which to print the header text
 *  @param uint16_t button_size: Size in which to print button text
 *  @param char * header: String of header text
 *  @param size_t header_length: Length of header string
 *  @param char * text: String of menu text
 *  @param size_t text_length: Length of menu text string
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
```

```
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int option: Returns option selected by user
 *
 *  @todo Add automatic word-wrapping
 *
 */

int draw_continue_cancel(uint16_t header_size, uint16_t button_size, char * header, size_t header_length, char * text,
size_t text_length, uint16_t menu_type,  Adafruit_TFTLCD * tft, TouchScreen * ts)
{


    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    char continue_text [] = " Continue ";
    char cancel_text [] = " Cancel ";

    char accept_text [] = " Accept ";
    char reject_text [] = " Reject ";

    byte i = 0;

    /// Draw blank screen
    (*tft).fillScreen(WHITE);


    /// Draw header bar and header text
    (*tft).fillRect(0, (header_size * 12), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(header_size);

    for(i = 0; i < header_length; i++)
    {
        (*tft).write(*(header + i));
    }


    /// Draw menu text
    (*tft).setCursor(10, (header_size * 12) + 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(button_size);

    for(i = 0; i < text_length; i++)
    {
        (*tft).write(*(text + i));
    }
```

```
    switch (menu_type)
    {

    case CONTINUE_CANCEL:
    {

        /// Create and draw "Continue" and "Cancel" buttons
        button button_continue(((*tft).width()/2) - (button_size * 60) - 20, (*tft).height() - (button_size * 20) - 10,
continue_text, sizeof(continue_text), button_size, tft, ts);
        button button_cancel(((*tft).width()/2) + 20, (*tft).height() - (button_size * 20) - 10, cancel_text,
sizeof(cancel_text), button_size, tft, ts);

        button_continue.draw(WHITE, BLACK, BLACK);
        button_cancel.draw(WHITE, BLACK, BLACK);


        /// Enter while loop and wait for user to make selection
        while(1)
        {

            /// This if statement executes if user selects "Continue"
            if(button_continue.get_status())
            {
                /// Return "Continue"
                return(CONTINUE);
            }

            /// This if statement executes if user selects "Cancel"
            if(button_cancel.get_status())
            {
                /// Return "Cancel"
                return(CANCEL);
            }
        }

        break;

    }

    case ACCEPT_REJECT:

    {

        /// Create and draw "Accept" and "Reject" buttons
        button button_accept(((*tft).width()/2) - (button_size * 60) - 20, (*tft).height() - (button_size * 20) - 10,
accept_text, sizeof(accept_text), button_size, tft, ts);
        button button_reject(((*tft).width()/2) + 20, (*tft).height() - (button_size * 20) - 10, reject_text,
sizeof(reject_text), button_size, tft, ts);
```

```
        button_accept.draw(WHITE, BLACK, BLACK);
        button_reject.draw(WHITE, BLACK, BLACK);


        /// Enter while loop and wait for user to make selection
        while(1)
        {

            /// This if statement executes if user selects "Accept"
            if(button_accept.get_status())
            {
                /// Return "Accept"
                return(ACCEPT);
            }

            /// This if statement executes if user selects "Reject"
            if(button_reject.get_status())
            {
                /// Return "Reject"
                return(REJECT);
            }
        }

    }

    }



    // Return zero... just because.
    return(0);



}
```

```
/**
 *
 *  This function draws the main menu of the OCFT system to the LCD. It waits for user to
 *  select an option, then returns the option to the calling function. Currently, the menu
 *  only has two options: "Run Test" and "Configure". Implementing more options shouldn't
 *  be too difficult, provided the menu is small enough to fit on one screen. If more options
 *  are needed, then the function will have to be modified to include support for scrolling,
 *  or else it will have to be split into two submenus, with the menus linked to each other
 *  through "Next" and "Previous" menu options.
 *
 *  @param uint16_t header_size: Size in which to print the header text
 *  @param uint16_t button_size: Size in which to print button text
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int option: Returns option selected by user
 *
 *  @todo Add support for other main menu options (if needed)
 ***********SECURE TEST MENU WAS ADDED
 */

int draw_main_menu(uint16_t header_size, uint16_t button_size, Adafruit_TFTLCD * tft, TouchScreen * ts)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    char header [] = "Main Menu";
    char button_1_text [] = " Run Test  ";
    char button_2_text [] = " Configure ";
    char button_3_text [] = " Secure Test ";

    byte i = 0;


    /// Draw blank screen
    (*tft).fillScreen(WHITE);


    /// Draw menu header bar and header text
    (*tft).fillRect(0, (header_size * 12), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
```

```
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(header_size);



    for(i = 0; i < sizeof(header); i++)
    {

        (*tft).write(*(header + i));

    }



    /// Create and draw menu option buttons.

    // Note the generic "button_x" naming convention. This makes it somewhat easier to add buttons to the function
later
    button button_1(10, ((header_size*12) + 10), button_1_text, sizeof(button_1_text), button_size, tft, ts);
    button button_2(10, ((header_size*12) + 10 + (button_size * 20)), button_2_text, sizeof(button_2_text),
button_size, tft, ts);
    button button_3(10, ((header_size*12) + 10 + (2 * button_size * 20)), button_3_text, sizeof(button_3_text),
button_size, tft, ts);



    button_1.draw(WHITE, BLACK, BLACK);
    button_2.draw(WHITE, BLACK, BLACK);
    button_3.draw(WHITE, BLACK, BLACK);


    /// Enter while loop, waiting for user to select an option
    while(1)
    {

        /// This if statement executes if the user selects option 1
        if(button_1.get_status())
        {
            /// Return user selection
            return(RUN_TEST);
        }

        /// This if statement executes if the user selects option 2
        if(button_2.get_status())
        {
            /// Return user selection
            return(CONFIGURE);
        }
        /// This if statement executes if the user selects option 3
        if(button_3.get_status())
        {
```

```
                /// Return user selection
                return(SECURE_TEST);
            }

        }


        /// This line of code will never see the light of day!
        return(0);

}
```

```
/**
 *
 *  This function draws the configuration menu for to the LCD. It then waits
 *  for user to select an option. Once the user has selected an option, it
 *  returns the selection to the calling function. As with other menu drawing
 *  functions, this function does not currently include and support for
 *  "scrolling." If more options are added, support for some kind of scrolling
 *  may have to be added.
 *
 *  @param uint16_t header_size: Size in which to print the header text
 *  @param uint16_t button_size: Size in which to print button text
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int option: Returns option selected by user
 *
 *  @todo Add support for other configuration menu options (if needed)
 *
 */

int draw_configure_menu(uint16_t header_size, uint16_t button_size, Adafruit_TFTLCD * tft, TouchScreen * ts)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-
```

```
    char header [] = "Configure";
    char button_1_text [] = "Sample Pressure Drop";
    char button_2_text [] = "Max Sample Time     ";
    char button_3_text [] = "Diagnostics         ";
    char button_4_text [] = "Back                ";

    byte i = 0;



    /// Draw blank screen
    (*tft).fillScreen(WHITE);


    /// Draw header bar and print header
    (*tft).fillRect(0, (header_size * 12), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(header_size);


    // Note that this for loop prints the header text. I used for loops before
    // I discovered that the "tft.print" function would work on strings, as
    // long as the string was hard coded into the function (i.e. not passed to
    // the function). In other words, this for loop could be replaced with
    //
    // (*tft).print("Configure");
    //
    // I just havent gotten around to doing so.
    for(i = 0; i < sizeof(header); i++)
    {
        (*tft).write(*(header + i));
    }


    /// Create and print button objects
    button button_1(10, ((header_size*12) + 10), button_1_text, sizeof(button_1_text), button_size, tft, ts);
    button button_2(10, ((header_size*12) + 10 + (1 * button_size * 20)), button_2_text, sizeof(button_2_text),
button_size, tft, ts);
    button button_3(10, ((header_size*12) + 10 + (2 * button_size * 20)), button_3_text, sizeof(button_3_text),
button_size, tft, ts);
    button button_4(10, ((header_size*12) + 10 + (3 * button_size * 20)), button_4_text, sizeof(button_4_text),
button_size, tft, ts);


    button_1.draw(WHITE, BLACK, BLACK);
    button_2.draw(WHITE, BLACK, BLACK);
    button_3.draw(WHITE, BLACK, BLACK);
```

```
    button_4.draw(WHITE, BLACK, BLACK);

    /// Enter while loop, wait for user input
    while(1)
    {

        /// This if statement executes if user selects option 1
        if(button_1.get_status())
        {
            /// Return user selection
            return(SET_SAMPLE_PRESSURE);
        }

        /// This if statement executes if user selects option 2
        if(button_2.get_status())
        {
            /// Return user selection
            return(SET_SAMPLE_TIME);
        }

        /// This if statement executes if user selects option 3
        if(button_3.get_status())
        {
            /// Return user selection
            return(DIAGNOSTICS);
        }

        /// This if statement executes if user selects option 4
        if(button_4.get_status())
        {
            /// Return user selection
            return(EXIT);
        }

    }

    // Why is this line of code here? It never gets executed.... but it's tradition.
    // Besides, what if the function somehow mysteriously managed to escape the while(1) loop?
    return(0);

}



/**
 *
 *  This function draws the SECURE TEST menu to the LCD. It then waits for the
 *  user to select an option. Note that as with every other menu draw function,
 *  this function has no support for "scrolling", so if buttons are added such
 *  that they don't all fit on one screen, scrolling of some kind will need to
```

```
 *  be added.
 *
 *  @param uint16_t header_size: Size in which to print the header text
 *  @param uint16_t button_size: Size in which to print button text
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int option: Returns option selected by user
 *
 *  @todo Add support for other configuration menu options (if needed)
 *
 */


int draw_secure_test_menu(uint16_t header_size, uint16_t button_size, Adafruit_TFTLCD * tft, TouchScreen * ts)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    char header [] = "Secure Test ";
    char button_1_text [] = "Suction Pressure";
    char button_2_text [] = "Secure Hardware ";
    char button_3_text [] = "Release Hardware ";
    char button_4_text [] = "Back              ";

    byte i = 0;



    /// Draw blank screen
    (*tft).fillScreen(WHITE);


    /// Draw header bar and print header
    (*tft).fillRect(0, (header_size * 12), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(header_size);



    for(i = 0; i < sizeof(header); i++)
    {
        (*tft).write(*(header + i));
    }


    /// Create and print button objects
    button button_1(10, ((header_size*12) + 10), button_1_text, sizeof(button_1_text), button_size, tft, ts);
```

```
    button button_2(10, ((header_size*12) + 10 + (1 * button_size * 20)), button_2_text, sizeof(button_2_text),
button_size, tft, ts);
    button button_3(10, ((header_size*12) + 10 + (2 * button_size * 20)), button_3_text, sizeof(button_3_text),
button_size, tft, ts);
    button button_4(10, ((header_size*12) + 10 + (3 * button_size * 20)), button_4_text, sizeof(button_4_text),
button_size, tft, ts);


    button_1.draw(WHITE, BLACK, BLACK);
    button_2.draw(WHITE, BLACK, BLACK);
    button_3.draw(WHITE, BLACK, BLACK);
    button_4.draw(WHITE, BLACK, BLACK);

    /// Enter while loop, wait for user input

   while(1)
    {

        /// This if statement executes if user selects option 1
        if(button_1.get_status())
        {
            /// Return user selection
            return(SET_SUCTION_PRESSURE);
        }

        /// This if statement executes if user selects option 2
        if(button_2.get_status())
        {
            /// Return user selection
            return(SECURE_HARDWARE);
        }

        /// This if statement executes if user selects option 3
        if(button_3.get_status())
        {
            /// Return user selection
            return(RELEASE_HARDWARE);
        }

        /// This if statement executes if user selects option 4
        if(button_4.get_status())
        {
            /// Return user selection
            return(EXIT);
        }

    }

    // Why is this line of code here? It never gets executed.... but it's tradition.
    // Besides, what if the function somehow mysteriously managed to escape the while(1) loop?
```

```
    return(0);

}
```

```
/**
 *
 *  This function draws the diagnostics menu to the LCD. It then waits for the
 *  user to select an option. Note that as with every other menu draw function,
 *  this function has no support for "scrolling", so if buttons are added such
 *  that they don't all fit on one screen, scrolling of some kind will need to
 *  be added.
 *
 *  @param uint16_t header_size: Size in which to print the header text
 *  @param uint16_t button_size: Size in which to print button text
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int option: Returns option selected by user
 *
 *  @todo Add support for other configuration menu options (if needed)
 *
 */

int draw_diagnostic_menu(uint16_t header_size, uint16_t button_size, Adafruit_TFTLCD * tft, TouchScreen * ts)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    char header [] = "Diagnostics";
    char button_1_text [] = "Measure System Volume";
    char button_2_text [] = "Back               ";


    byte i = 0;


    /// Draw blank screen
```

```
    (*tft).fillScreen(WHITE);


    /// Draw header bar and print header
    (*tft).fillRect(0, (header_size * 12), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(header_size);

    // Another unnescessary for loop
    for(i = 0; i < sizeof(header); i++)
    {

        (*tft).write(*(header + i));

    }

    /// Create and print menu option buttons
    button button_1(10, ((header_size*12) + 10), button_1_text, sizeof(button_1_text), button_size, tft, ts);
    button button_2(10, ((header_size*12) + 10 + (1 * button_size * 20)), button_2_text, sizeof(button_2_text),
button_size, tft, ts);

    button_1.draw(WHITE, BLACK, BLACK);
    button_2.draw(WHITE, BLACK, BLACK);

    /// Enter while loop waiting for user input
    while(1)
    {

        /// This if statement executes if user selects option 1
        if(button_1.get_status())
        {
            /// Return user selection
            return(CALIB_SYSTEM_VOLUME);
        }

        /// This if statement executes if user selects option 2
        if(button_2.get_status())
        {
            /// Return user selection
            return(EXIT);
        }
    }

    // Another line of code that will never run
    return(0);

}
```

```
/**
 *
 *  This function draws a "Yes/No" menu to the LCD, and waits for user input. Function
 *  allows for the text of the menu to be determined by the calling function. However,
 *  the function does not include support for rejecting text that is too large. So if
 *  the calling function sends text that is too big for the screen, the text will run
 *  over into the button area. Also, note that function includes no support for
 *  word-wrapping. Currently it is up to the calling function to ensure that the text
 *  has newline characters in place where needed for proper word wrapping. (Menu will
 *  TEXT wrap, but not word wrap.)
 *
 *  @param uint16_t header_size: Size in which to print the header text
 *  @param uint16_t button_size: Size in which to print button text
 *  @param char * header: String of header text
 *  @param size_t header_length: Length of header string
 *  @param char * text: String of menu text
 *  @param size_t text_length: Length of menu text string
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int option: Returns option selected by user
 *
 *  @todo Add automatic word-wrapping
 *
 */
```

```
int draw_yes_no_menu (uint16_t header_size, uint16_t button_size, char * header, size_t header_length, char * text,
size_t text_length, Adafruit_TFTLCD * tft, TouchScreen * ts)
{

    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    char yes_text [] = " Yes ";
    char no_text [] = " No  ";

    byte i = 0;


    /// Draw blank screen
    (*tft).fillScreen(WHITE);


    /// Draw header bar and header
    (*tft).fillRect(0, (header_size * 12), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(header_size);

    // Note, this for loop IS needed, since the string was passed to the function from the calling function
    for(i = 0; i < header_length; i++)
    {
        (*tft).write(*(header + i));
    }


    /// Print menu text
    (*tft).setCursor(10, (header_size * 12) + 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(button_size);

    // This for loop is also nescessary
    for(i = 0; i < text_length; i++)
    {

        (*tft).write(*(text + i));

    }

    /// Create and draw "Yes" and "No" buttons
    button button_yes(((*tft).width()/2) - (button_size * 35) - 20, (*tft).height() - (button_size * 20) - 10, yes_text,
sizeof(yes_text), button_size, tft, ts);
    button button_no(((*tft).width()/2) + 20, (*tft).height() - (button_size * 20) - 10, no_text, sizeof(no_text),
button_size, tft, ts);
```

```
    button_yes.draw(WHITE, BLACK, BLACK);
    button_no.draw(WHITE, BLACK, BLACK);


    /// Enter while loop and wait for user input
    while(1)
    {

        /// This if statement executes if user selects "Yes"
        if(button_yes.get_status())
        {
            /// Return YES
            return(YES);
        }

        /// This if statement executes if user selects "No"
        if(button_no.get_status())
        {
            /// Return NO
            return(NO);
        }
    }


    // Yes, I include the
    //
    // return(0);
    //
    // statement, regardless of whether it's needed. You should too.
    return(0);

}
```

```
/**
```

```
 *
 *  This function draws a "Continue"/"Cancel" menu to the LCD. It then
 *  waits for user input.  Function allows for the text of the menu to
 *  be determined by the calling function. However, the function does not
 *  include support for rejecting text that is too large. So if the
 *  calling function sends text that is too big for the screen, the text
 *  will run over into the button area. Also, note that function includes
 *  no support for word-wrapping. Currently it is up to the calling
 *  function to ensure that the text has newline characters in place where
 *  needed for proper word wrapping. (Menu will TEXT wrap, but not word
 *  wrap.)
 *
 *  @param uint16_t header_size: Size in which to print the header text
 *  @param uint16_t button_size: Size in which to print button text
 *  @param char * header: String of header text
 *  @param size_t header_length: Length of header string
 *  @param char * text: String of menu text
 *  @param size_t text_length: Length of menu text string
 *  @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 *  @param Touchscreen * ts: Pointer to touchscreen interface object
 *
 *  @retval int option: Returns option selected by user
 *
 *  @todo Add automatic word-wrapping
 *
 */

int draw_continue_cancel(uint16_t header_size, uint16_t button_size, char * header, size_t header_length, char * text,
size_t text_length, Adafruit_TFTLCD * tft, TouchScreen * ts)
{


    /// -PSUEDO CODE INCLUDED IN INLINE COMMENTS BELOW-

    char continue_text [] = " Continue ";
    char cancel_text [] = " Cancel ";

    byte i = 0;

    /// Draw blank screen
    (*tft).fillScreen(WHITE);


    /// Draw header bar and header text
    (*tft).fillRect(0, (header_size * 12), (*tft).width(), 5, BLACK);

    (*tft).setCursor(10, 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(header_size);
```

```
    for(i = 0; i < header_length; i++)
    {
        (*tft).write(*(header + i));
    }


    /// Draw menu text
    (*tft).setCursor(10, (header_size * 12) + 10);
    (*tft).setTextColor(BLACK);
    (*tft).setTextSize(button_size);

    for(i = 0; i < text_length; i++)
    {
        (*tft).write(*(text + i));
    }

    /// Create and draw "Continue" and "Cancel" buttons
    button button_continue(((*tft).width()/2) - (button_size * 60) - 20, (*tft).height() - (button_size * 20) - 10,
continue_text, sizeof(continue_text), button_size, tft, ts);
    button button_cancel(((*tft).width()/2) + 20, (*tft).height() - (button_size * 20) - 10, cancel_text, sizeof(cancel_text),
button_size, tft, ts);

    button_continue.draw(WHITE, BLACK, BLACK);
    button_cancel.draw(WHITE, BLACK, BLACK);


    /// Enter while loop and wait for user to make selection
    while(1)
    {

        /// This if statement executes if user selects "Continue"
        if(button_continue.get_status())
        {
            /// Return "Continue"
            return(CONTINUE);
        }

        /// This if statement executes if user selects "Cancel"
        if(button_cancel.get_status())
        {
            /// Return "Cancel"
            return(CANCEL);
        }
    }


    // Return zero... just because.
    return(0);

}
```

```
/**
 * This function prints a number to the touchscreen in scientific format.
 *
 * @param Adafruit_TFTLCD * tft: Pointer to LCD display object
 * @param double value: Value to be printed
 * @param int places: Number of decimal places to which to print value
 *
 * @retval int 0: return 0 when function exits
 *
 */

int print_scientific(Adafruit_TFTLCD * tft, double value, int places)
{

	int i = 0;
	char negative = 0;


	/// Store sign information of number
	if(value < 0)
	{
		negative = 1;
	}

	/// Set number to absolute value for conversion process
	value = abs(value);



	/// This if statement handles the number if it is greater than 10
	if (abs(value) > 10.0)
	{

		/// While loop divides by 10 until number is less than 10
		while (abs(value) > 10.0)
		{

			i++;

			value = value/10.0;

		}
```

```
        /// Revert to original sign
        if(negative)
        {
            value = -value;
        }


        /// Print value with power
        (*tft).print(value, places);

        (*tft).print("e+");

        (*tft).print(i);

        return (0);

    }


    /// This if statement handles the number if it is less than 1
    if (abs(value) < 1.0)
    {

        /// While loop multiplies number by 10 until number is greater than 1
        while (abs(value) < 1.0)
        {

            i++;

            value = value * 10.0;


        }

        /// Revert to original sign
        if(negative)
        {
            value = -value;
        }


        /// Print value with power
        (*tft).print(value, places);

        (*tft).print("e-");

        (*tft).print(i);

        return (0);
```

```
    }


    /// If number is greater than 1 and less than 10, the function simply prints the value with the power.

    if(negative)
    {
        value = -value;
    }

    (*tft).print(value, places);

    (*tft).print("e+");

    (*tft).print(0);

    return (0);

}
```

```
//@}

\
/**
 *  Project: On-Canister Filter Tester
 *
 *  @file functions.h
 *  @version 0.1.11
 *  @author Joel Runnels
 *  @date 2013-08-14
 *
 *  @brief  This file contains function prototypes for functions used in OCFT,
 *  and some structure definitions
 *
 *  @details
 *  Functions grouped by type, then sorted alphabetically. Functions stored by the
 *  following categories:
 *
 *  - Computational Functions
 *
 *    This category contains functions which perform some sort of computational
 *    function. These functions generally do not involve any instrument or
 *    hardware interfacing, and are designed to be as general as possible.
```

```
*    Examples include trapezoidal integration function and the volume computation
*    function.
*
*
*  - Hardware and Instrumentation Interface Functions
*
*    This category contains functions which interface directly with hardware
*    and instrumentation. These functions usually involve interfacing directly
*    with instrument pins, and return values of sensor readings if available.
*
*  - EEPROM Interface Functions
*
*    This category contains functions which read or write to microcontroller
*    onboard EEPROM. Functions are specifically designed to read or write a
*    particular value to EEPROM. Examples include the function which reads the
*    max sample time from EEPROM.
*
*  - Menu Logic Functions
*
*    This category contains functions which handle flow control and logic of
*    the programming. These functions are generally the highest-level functions
*    in the program which pass user input and output to calling functions.
*
*  - Process Functions
*
*    This category contains functions which control higher-level processes
*    onboard the instrument. Examples of functions in this category include
*    the functions responsible for measuring volume of a container and running
*    filter tests. These functions often involve controlling hardware,
*    interfacing with instruments, and performing computations. These functions
*    usually call multiple lower-level functions during the course of their
*    execution. These functions also interface with the touchscreen to display
*    current data as needed.
*
*  - Touchscreen Interface Functions
*
*    This category contains functions which are responsible solely for
*    touchscreen interfacing, both displaying data to touchscreen and receiving
*    user input from touchscreen. These functions often pass user input directly
*    to the logic flow control functions of the software.
*
*
*  @todo Continue documentation and commenting effort
*
*/


#ifndef FUNCTIONS_H
#define FUNCTIONS_H
```

```
#include <Arduino.h>
#include <Adafruit_GFX.h>    // Core graphics library
#include <Adafruit_TFTLCD.h> // Hardware-specific library
#include <TouchScreen.h>
#include "buttons.h"
#include "define.h"



/***************************/
/*  Structure Definitions  */
/***************************/


/**
 *
 * @brief Simple data structure for storing pressure and correlated volume readings
 *
 */

struct pv_data
{
      byte pressure; ///< Pressure, usually in inches of water
      double volume; ///< Volume, usually in cm^3
};



/**
 *
 * @brief Data structure for compacting data to pass to scroll menu
 *
 * This structure is only used when calling the draw_scroll_menu function. Note
 * that if more parameters need to be added to the scroll menu, they can be
 * included in this structure.
 *
 */

struct scroll_menu_input
{
      uint16_t header_size; ///< Size of header text
      uint16_t button_size; ///< Size of button text
      char * scroll_title; ///< Scroll menu title string
      size_t scroll_size; ///< Length of scroll title string
      char * units; ///< Units sting
      size_t unit_size; ///< Length of unit string
      double val_init; ///< Initial variable value
      double val_min; ///< Minimum value for variable
      double val_max; ///< Maximum value for variable
```

```
    double warning_min; ///< Value at which menu gives warning
    double warning_max; ///< Value at which menu gives warning
    double increment; ///< Amount by which to increment variable up or down
};
```

```
/*************************/
/*  FUNCTION PROTOTYPES   */
/*************************/
```

```
/* Menu Logic Functions */
/**
 * @anchor menu_logic_functions
 * @name Menu Logic Functions
 *
 *  This category contains functions which handle flow control and logic of
 *  the programming. These functions are generally the highest-level functions
 *  in the program which pass user input and output to calling functions.
 */
//@{
int main_menu(Adafruit_TFTLCD *, TouchScreen *);
int configure_menu(Adafruit_TFTLCD *, TouchScreen *);
int diagnostics_menu(Adafruit_TFTLCD *, TouchScreen *);
int secure_test_menu(Adafruit_TFTLCD *, TouchScreen *);
//@}
```

```
/* Computational Functions */
/**
 * @name Computational Functions
```

```
 *
 *  This category contains functions which perform some sort of computational
 *  function. These functions generally do not involve any instrument or
 *  hardware interfacing, and are designed to be as general as possible.
 *  Examples include trapezoidal integration function and the volume computation
 *  function.
 */
//@{
double compute_mass_flow_in(double, double, double, double);
double compute_volume(double, double, double, double);
double derivative(double, double, double);
double trap_integrate(double, double, double);
//@}
```

```
/* Hardware and Instrument Interface Functions */
/**
 * @name Hardware and Instrument Interface Functions
 *
 *  This category contains functions which interface directly with hardware
 *  and instrumentation. These functions usually involve interfacing directly
 *  with instrument pins, and return values of sensor readings if available.
 */
//@{
double read_mass_flow(int, char);
double read_pressure(int, char, byte rolling_average = 1);
//@}
```

```
/* EEPROM Interface Functions */
/**
 * @name EEPROM Interface Functions
```

```
 *
 *  This category contains functions which read or write to microcontroller
 *  onboard EEPROM. Functions are specifically designed to read or write a
 *  particular value to EEPROM. Examples include the function which reads the
 *  max sample time from EEPROM.
 */
//@{
byte settings_read_time(void);
byte settings_write_int_volume(pv_data *);
byte settings_write_pressure(double);
byte settings_write_pressure1(double);
byte settings_write_time(byte);

double settings_read_pressure(void);
double settings_read_pressure1(void);
double settings_read_int_volume(double);
//@}
```

```
/* Process Functions */
/**
 * @anchor process_functions
 * @name Process Functions
 *
 *  This category contains functions which control higher-level processes
 *  onboard the instrument. Examples of functions in this category include
 *  the functions responsible for measuring volume of a container and running
 *  filter tests. These functions often involve controlling hardware,
 *  interfacing with instruments, and performing computations. These functions
 *  usually call multiple lower-level functions during the course of their
 *  execution. These functions also interface with the touchscreen to display
 *  current data as needed.
 */
//@{
int calib_system_volume_ts(Adafruit_TFTLCD *, TouchScreen *);
int leak_check_ts(double *, double *, double *, double *, byte *, byte *, Adafruit_TFTLCD *, TouchScreen *, byte);
int measure_volume_ts(double, int, double, double *, byte *, byte *, byte, Adafruit_TFTLCD *, button *, double *);
int run_test_ts(Adafruit_TFTLCD *, TouchScreen *);
int run_secure_hardware_ts(Adafruit_TFTLCD *, TouchScreen *);
```

```
int run_release_hardware_ts(Adafruit_TFTLCD *, TouchScreen *);
int secure_hardware_ts(byte *,Adafruit_TFTLCD *, button *, double *);
//@}




/* Touchscreen Interface Functions */

/**
 * @name Touchscreen Interface Functions
 *
 *  This category contains functions which are responsible solely for
 *  touchscreen interfacing, both displaying data to touchscreen and receiving
 *  user input from touchscreen. These functions often pass user input directly
 *  to the logic flow control functions of the software.
 */
//@{
double draw_scroll_menu (scroll_menu_input, Adafruit_TFTLCD *, TouchScreen *);

int cycle_color(uint16_t *);
int draw_configure_menu(uint16_t, uint16_t, Adafruit_TFTLCD *, TouchScreen *);
int draw_continue_cancel(uint16_t, uint16_t, char *, size_t, char *, size_t, uint16_t, Adafruit_TFTLCD *, TouchScreen *);
int draw_diagnostic_menu(uint16_t, uint16_t, Adafruit_TFTLCD *, TouchScreen *);
int draw_main_menu(uint16_t, uint16_t, Adafruit_TFTLCD *, TouchScreen *);
int draw_secure_test_menu(uint16_t, uint16_t, Adafruit_TFTLCD *, TouchScreen *);
int draw_yes_no_menu (uint16_t, uint16_t, char *, size_t, char *, size_t, Adafruit_TFTLCD *, TouchScreen *);
int print_scientific(Adafruit_TFTLCD *, double, int places = 2);

//@}




#endif
```