TO: The Honorable Judge Michael Patrick King, P.J.A.D, retired and Special Master

Report On Behalf of The Defendants:

JANE H. CHUN, DARIA L. DE CICCO, JAMES R. HAUSLER, ANGEL MIRALDA, JEFFREY R. WOOD, ANTHONY ANZANO, MEHMET DEMIRELLI, RAJ DESIA, JEFFREY LOCASTRO, PETER LIEBERWIRTH, JEFFREY LING, HUSSAIN NAWAZ, FREDERICK OGBUTOR, PETER PIASECKI, LARA SLATER, CHRISTOPHER SALKOWITZ, ELINA TIRADO, DAVID WALKER, DAVID WHITMAN and JAIRO J. YATACO

Prepared By:
John J. Wisniewski
Base One Technologies
10 S. Division Street
New Rochelle, NY 10805

TABLE OF CONTENTS

SECTION I OVERVIEW OF KEY FINDINGS

Key Findings- Testing the Alcotest 7110 Mk III Source Code uncovered 24 major defects. For the purposes of this overview, we have identified 9 defects with the greatest impact on the instrument test results, and the validity of those tests.

All Results/Findings for the Alcotest 7110, along with detailed explanations are found in Section IV of this Report. Conclusions are in Section V.

1. **Alcotest Software Would Not Pass U.S. Industry Standards for Software Development and Testing**
2. **Proof of Incomplete Software Testing**
3. **Catastrophic Error Detection Is Disabled**
4. **Implemented Design Lacks Positive Feedback**
5. **Diagnostics Adjust or Substitute Data Readings**
6. **Flow Measurements Adjust and Substitute**
7. **Error Detection Logic**
8. **Software Does Not Insulate/ Protect Modules or Data**
9. **Timing Problems**

Of The 24 Major Defects Uncovered In The 7110Mk III Source Code, These 9 Defects Greatly Impact The Validity Of This Instruments Test Results

1. Alcotest Software Would Not Pass U.S. Industry Standards for Software Development and Testing - It is clear that the Alcotest software would not pass development standards and testing for the U.S. Government, the U.S. Military, the Federal Aviation Administration or the Federal Drug Administration, as well as commercial standards used in devices for public safety. Here, ***standards require*** the source code be available for audit by the approving agencies. Industry Standards exist and are required when work is performed for government agencies. The quality, accuracy and reliability of work done to industry standards is consistently superior to products developed without standards.

2. Proof of Incomplete Software Testing – The claim that the Alcotest software has been tested thoroughly over some amount of time ignores known computer science principles. The only rigid standard of testing software is to ensure that all code has been executed or all software paths have been examined. Following that path of software is fruitless, because it is easily demonstrated that it is *not possible* to test everything. The Alcotest software contains over

45,000 lines of code (computer instructions), of which 3200 lines of code are designed to make decisions. Each decision can change the software execution path, and each decision affects subsequent decisions. This situation is not unique to the Alcotest software, as even software programs developed under industry standards would still not be able to test all the decision paths. *However, the lack of use of industry coding standards prevents the testing of critical paths in the Alcotest software, and prevents removal of defects during the requirements, design, and coding process.*

3. Catastrophic Error Detection Is Disabled – The code has disabled capabilities in the processor that detect catastrophic problems when instructions are executed with regard to invalid and/or illegal data values or with corrupted instructions. *Turning off these safeguards means as these conditions are encountered, the machine produces unpredictable results.*

4. Implemented Design Lacks Positive Feedback – When the processor changes the state of a device, such as a motor or valve, the motor or valve may fail to respond because of a malfunction or other cause. The Alcotest does not have circuitry, sensors, and verification software components to verify the controlled device. *This means that the software assumes the change in state is always correct, but it cannot verify the action.* Analogy: when a pilot lowers the landing gear on a commercial plane, the systems on the plane sense and confirm that the landing gear lowered. Without this confirmation the pilot could crash the plane, because of the assumption that the gear lowered and it did not.

5. Diagnostics Adjust/Substitute Data Readings - The diagnostic routines for the Analog to Digital (A/D) Converters are performed during the data measurement cycles of the Alcotest (breath measurement, purge, etc.), and the diagnostics are performed on the measurement/ data values taken. Diagnostic routines in other similar software applications are executed during a separate period from the measurement periods. *If a diagnostic fails, the Alcotest will substitute arbitrary "canned" data values for the measured device, thereby affecting the breath measurements.*

6. Flow Measurements Adjusted/Substituted - The software presumes at the beginning of the measurement that no airflow is going through the machine. The software then takes an airflow reading, and whatever number is read, this value is assumed to be the "zero value" or baseline. No quality check or reasonableness test is done on this baseline measurement. (Is 100 good? 500?) Subsequent calculations are compared against this baseline measurement, and the difference is considered the change in airflow. So, if the baseline value is 500, and the next reading is 400, then the airflow measurement is –100, or the air is flowing in reverse. The software logic also detects data measurement failures, and if these occur, then the software substitutes the last known good baseline value, even if it occurred a long time ago. *When this calibrated value is either corrupted or is initially computed incorrectly, measurements made by the machine prior to the new power down/power up sequence, are defective.*

7. Error Detection Logic - The software design detects measurement errors, but ignores/ suppresses these error messages unless they occur a large number of consecutive times. *For example, if an error has to occur 32 times before reporting, then the error could occur 31 times, then appear within range once, then appear 31 times, etc., and be considered a properly working device.*

8. Software Does Not Insulate/Protect Modules or Data – Embedded processors do not possess the ability to prevent inadvertent change to data locations in memory or to instructions. The Alcotest software relies heavily on "global" data variables and global functions or subroutines, *which means any incorrectly coded or modified functions can inadvertently modify a data value not part of that routine's sphere of influence.*

9. Timing Problems - The design of the code is to run in timed units of 8.192 milliseconds, by means of an interrupt signal to a handler, which then signals the main program control that it can continue to the next segment. The interrupt goes off every 8.192 ms, not 8.192 ms from the latest request for a time delay. *This means the analysis of the mainline code dependent on the interrupt timer operation will reveal that the actual timings will vary widely, and be inconsistent.*

Analogy: You have an “alarm clock” that has an hour hand, minute hand, and second hand, and the hands move in a stepping motion from one tic to another. This clock chimes every minute on the minute (when the second hand reaches 12). If you want this clock to time a minute and tell you when the minute happens, and the second hand is at 5 seconds when you submit the request, then the alarm will go off 55 seconds later, which is close enough. If instead, you request the minute timing and the second hand is at 55 seconds, then your “minute timer” in that instance would be only 5 seconds long.

Common-sense practice is to write interrupt handler software as small units of code that operate quickly, and for a short period of time. *The Alcotest external interrupt routines are very lengthy and are written in C instead of assembly language, which will increase the execution time and the memory used. There is a possibility that the interrupt handler can operate on an interrupt, take too long and miss the next one, which could produce incorrect and unexpected results.*

SECTION II OUR ASSUMPTIONS

1. We assumed that the binary image given actually represented the code in the Alcotest machine.

2. We did not assume the code supplied produced the binary image supplied, and we set out to verify that the code produced the image. If the image was verified, then we could enumerate which supplied modules contributed to the image, and which modules were superfluous. We assumed that superfluous code was part of optional features or part of new work in progress; this is common in software development.

3. We assumed that schematics, datasheets, and manuals would be supplied with the source code, as well as scripts, compiling instructions, “make files”, compilers, and other software tools.

4. We assumed we would find problems in areas of the code common to embedded systems. These include timing problems, problems in handling interrupts, rogue execution of instructions, memory management problems, array and vector processing, device control problems, and memory banking (shared memory) management.

5. We assumed there could be inconsistencies in data handling with respect to unit conversions, argument passing between functions, and data acquisition.

6. We assumed comments placed in the source code, especially those pointing out issues and problems, to be worthy of investigation. We treated such comments with skepticism until the comments were proven reasonably correct or relevant.

Additional Assumptions by Base One

7. We assumed the source code transmitted from Drager was complete and the version of source code to be verified as that represented in NJ version 3.11 firmware.
8. We assumed the compilers and tools transmitted to John Wisniewski were correct. These instructions proved to be defective. However, based on his years of experience with similar tools John was able to surmount the problem.
9. We assumed the translation tools would provide an adequate translation from German to English. For critical interpretations, we used a German-speaking translator to accurately translate comments for analysis and comparison to the source code.

SECTION III ANALYSIS PROCESS AND TOOLS

The initial approach used to analyze the Alcotest 7110 Mk III source code was to look for identifying information indicating which software coding development standards Drager used in the development of the software. Typically, programmers insert comments into the code such as "coded in this manner to satisfy ISO 9000"[1] as such indicators.

It is important to emphasize that these standards are not "inspection methods" or "checklists" to reverse-engineer code. The standards are development tools, which improve software design and coding by encouraging standardized coding and design.

We conformed to the following steps in analyzing the source code provided by Drager.

1. Compiled[2] and linked[3] the source code[4] provided.
2. When built, compared the binary file produced against the image file[5] provided.
3. Used the makefile[6] to identify source files[7] supplied that were not used in the build[8], and therefore could be ignored.

---

[1] **ISO 9000** is a family of standards for quality management systems. ISO 9000 is maintained by ISO, the International Organization for Standardization and is administered by accreditation and certification bodies.

[2] Compiler, a computer program (or set of programs) that translates source code written in a computer language into another computer language (the target language)

[3] Linker, a program that takes one or more objects generated by compilers and assembles them into a single executable program

[4] Any sequence of statements and/or declarations written in some human-readable computer programming language

[5] Image file formats provide a standardized method of organizing and storing image data

[6] In software development, **make** is a utility for automatically building large applications. Files specifying instructions for *make* are called **Makefiles**. *Make* is an expert system which tracks which files have changed since the last time the project was built and invokes the compiler on only those source code files and their dependencies.

[7] A place from which data is taken. Many computer commands involve moving data. The place from which the data is moved is called the *source*, whereas the place it is moved to is called the destination or *target*.

[8] To program, or write lines of code.

4. Found and started with the "main" function.
5. Next, identified function calls to establish the calling tree[9] down to the lowest level.
6. Uncovered the interrupt handlers[10] and determined how they were used.
7. Used the following tools
    a. "Understand" (a C analysis program) to verify the calling tree and identify functions
    b. LINT[11] to identify syntax[12] and coding[13] problems.
    c. Reviewed the output of the linker to look at the code architecture[14].
    d. Software language translators to translate the German used for variable names and annotating comments.
8. Continual, recursive code review.
9. Investigated source code comments.
10. Investigated and tried to verify areas in the code where problems typically occur, particularly since industry accepted coding standards were not used.

---

[9] In computer science, a **tree** is a widely-used data structure that emulates a tree structure with a set of linked nodes.

[10] A signal informing a program that an event has occurred. When a program receives an interrupt signal, it takes a specified action (which can be to ignore the signal). Interrupt signals can cause a program to suspend itself temporarily to service the interrupt. Each type of software interrupt is associated with an *interrupt handler* -- a routine that takes control when the interrupt occurs.

[11] In computer programming, **lint** was the original name given to a particular tool that flagged suspicious and non-portable constructs (i.e., likely to be bugs) in C language source code. The term is now applied generically to tools that flag suspicious usage in software written in any computer language.

[12] Refers to the spelling and grammar of a programming language. Computers are inflexible machines that understand what you type only if you type it in the exact form that the computer expects. The expected form is called the syntax.

[13] Written computer instructions. Code can appear in a variety of forms. The code that a programmer writes is called *source code*. After it has been compiled, it is called *object code*. Code that is ready to run is called *executable code* or *machine code*

[14] The architecture of a system always defines its broad outlines, and may define precise mechanisms as well.

Tools Used:

1. Lint – code checker from UNIX[15]/Linux[16]/open source[17]
2. Microsoft Visual C++ Development Environment and Compiler
3. Borland C++ 4.52
4. IAR embedded C compiler
5. "Understand" C code analyzer
6. Source Format X
7. Easy Translator
8. GE Trans translator
9. Alta Vista Babelfish
10. Beyond Compare v. 2.1.2

[15] A multi-user, multitasking operating system widely used as the master control program in computer workstations and servers. UNIX is written in C.

[16] A version of UNIX. Linux is freeware.

[17] Generically, open source refers to a program in which the source code is available to the general public for use and/or modification from its original design free of charge, i.e., open.

SECTION IV FINDINGS / RESULTS

Alcotest 7110 Mk III Source Code Results

Characteristics of Alcotest Software

The program presented shows ample evidence of incomplete design, incomplete verification of design, and incomplete "white box" and "black box" testing. Therefore the software has to be considered unreliable and untested, and in several cases it does not meet stated requirements. The source code supplied has creation dates and modification dates from 1993 to 1997, but the coding architecture, style, organization, and modification documentation (audit trail) more closely resemble the software principles used in the 1970's and 1980's. The planning and documentation of the design is haphazard. Sections of the original code and modified code show evidence of using an experimental approach to coding, or use what is best described as the "trial and error" method. Several sections are marked as "temporary, for now". Other sections were added to existing modules or inserted in a code stream, leading to a patchwork design and coding style.

The commonly used software verification principle (testing) for this coding/development era is to execute all software logic paths and verify the operation/output is correct for each path. Unfortunately, complex software in the Alcotest, tested in this manner cannot be fully tested, by mathematical proof. Testing of complex systems cannot test all logic paths. As the number of logic paths increases, the number of tests required to test all the combinations of the paths rises exponentially. Therefore, the time required to test all paths is not practical to achieve. The premise that the software is reliable is therefore not based on evidence, data, experimental tests

or procedures, but on an opinion, arbitrary decision or even a desired release date. This topic is explained more fully in the section entitled *Proof of Incomplete Software Testing"* on page 15.

The only current alternative to this testing method (except for no testing at all), is to use the software development life-cycle concept. This concept is governed by one of the nationally and internationally recognized development standards to prevent defects from entering the software during the design process, and to find and eliminate more defects as the software is coded, tested, and released to the field.

This concept of software development using standards requires extensive and meticulous supporting data, and notations in source files, and a configuration management system. None of this methodology is evident in the Alcotest code. Further, the decision method of how to allocate the architecture and assignment of tasks does not match any of the software standards. This further substantiates that software development standards were not used to verify or test the software, including the ISO 9000 family of standards.

None of the segments of source code is marked "proprietary", "company sensitive", "confidential", etc. to alert employees as to inadvertent exposure. Code that could not be deemed proprietary (such as calculating the sum of a group of variables, a universal algorithm) is not clearly separated from proprietary code. Code purchased from vendors or in the public domain or open source is also not recognized.

The code consists mostly of general algorithms arranged in a manner to implement the breath testing sequence. That is, the code is not really unique or proprietary.

The premise that a development house has to treat code as proprietary is an obsolete idea. Marking code as proprietary and hiding it from public view will not prevent the development of a similar competitive device. The code design is driven by the devices it interacts with, and datasheets and application notes for the hardware will describe how software interacts with it. A proprietary designation for the software may be a red herring to competitors, dissuading them from competing with the device, but only for psychological reasons.

The code is not arranged in a "core" section, "custom" section, or any other additional division or organization. It is possible that a thread segment might be considered a "core" or untouchable segment, yet there is no indication where that segment either begins or ends, nor is anything identifiable regarding "custom" sections. A new programmer working for Drager would have no idea which areas of code he/she could modify, and which was untouchable.

Several sections that conceivably could be considered core, such as the interrupt handlers and interrupt vector definitions, appear nonetheless to have been changed after the code came to the United States. This is based on most of the code in a module containing German comments, and German variables and identifiers, then abruptly the coding changes to English for a short time, and then returns to German.

At least three different programmers created the Alcotest source files, based on information in the headers and differing coding styles. The reading of the code is difficult, crowded, and "choppy". It appears Drager did not enforce an internal standard to code the modules in a uniform way. This means that programmers did not layout their code in a uniform way. The software will still execute as written, but it will be more difficult to find defects, or find where functions are located.

Most of the time, a "header" section identifies the module, the programmer, and the date, but the rest is free flowing, and several files are even missing the header. The three or more programmers had distinct styles, but none of the styles is easily readable. Sections of the header are empty, such as the last time the code was modified, and the author of the change.

Approval by Other Standards

It is clear that, as submitted, the Alcotest software would not pass development standards and testing for the U.S. Government or Military. It would fail software standards for the Federal Aviation Administration (FAA) and Federal Drug Administration (FDA), as well as commercial standards used in devices for public safety.

This means the Alcotest would not be considered for military applications such as analyzing breath alcohol for fighter pilots. If the FAA imposed mandatory alcohol testing for all commercial pilots, the Alcotest would be rejected based upon the FAA safety and software standards.

These standards require the source code to be available for audit by the approving agencies. Even in the competitive commercial aircraft world, software has not been copied by competitors and has been protected by these agencies.

Proof of Incomplete Software Testing

The Alcotest software is a complicated software program, with over 45,000 lines of code (computer instructions), and over 3,200 of these lines are instructions which make decisions, meaning the software execution path will change one way or another based on these decisions[18].

The calculation for traversing the decision paths once is $3200^2 = 10{,}240{,}000$. If we postulate one second to test one branch, then this test would take 118.5 days to test just once. This demonstrates that the exhaustive testing method is too costly and error prone. Therefore, the software has not been thoroughly tested, if "tested" means that all code has been executed or all software paths have been examined.

The premise that the Alcotest software has been tested thoroughly because failures have not occurred over some arbitrary period purposely ignores unchecked paths, because testing stopped before all paths could be tested.

It should be mentioned here that programs developed under the industry standards mentioned would still not be able to test all the paths. However, these standards help identify and rank critical paths in the software to test, enforce coding standards that require tests like array bounds

[18] Note: a figure of 57,000 lines of code or greater has been quoted as the number to use for the Alcotest, but this number includes the source modules that are present in the directory but not used, and also includes the code for the communication program on the PC, which does not run on the Alcotest.

checking, and eliminate defects introduced into the code by the development process. These standards are currently the best available method.

Multiple Measurements

The ten-percent difference comparison of successive tests is done in units of ug/l, not %BAC. There is a comment in that area of the code stating "this conversion to %BAC needs to be done", but the conversion is not done.

Readings not Averaged Correctly

When the software takes a series of readings, it first averages the first two readings. Then, it averages the third reading with the average just computed. Then the fourth reading is averaged with the new average, and so on. There is no comment or note detailing a reason for this calculation, which would cause the first reading to have more weight than successive readings. Nonetheless, the comments say that the values should be averaged, and they are not.

Results Limited to Small, Discrete Values

The A/D converters measuring the IR readings and the fuel cell readings can produce values between 0 and 4095. However, the software divides the final average(s) by 256, meaning the final result can only have 16 values to represent the five-volt range (or less), or, represent the range of alcohol readings possible. This is a loss of precision in the data; of a possible twelve bits of information, only four bits are used. Further, because of an attribute in the IR calculations, the result value is further divided in half. This means that only 8 values are possible for the IR detection, and this is compared against the 16 values of the fuel cell.

Catastrophic Error Detection is Disabled

An interrupt that detects that the microprocessor is trying to execute an illegal instruction is disabled, meaning that the Alcotest software could appear to run correctly while executing wild branches or invalid code for a period of time. Other interrupts ignored are the Computer Operating Property (a watchdog timer), and the Software Interrupt.

Implemented Design Lacks Positive Feedback

The software controls electrical lines, which switch devices on and off, such as an air pump, infrared source, etc. The design does not provide a monitoring sensory line (loop back) for the software to detect that the device state actually changed. This means that the software assumes the change in state is always correct, but it cannot verify the action.

Diagnostics Adjust/Substitute Data Readings

The diagnostic routines for the Analog to Digital (A/D) Converters will substitute arbitrary, favorable readings for the measured device if the measurement is out of range, either too high or too low. The values will be forced to a high or low limit, respectively. This error condition is suppressed unless it occurs frequently enough (see below).

Flow Measurements Adjusted/Substituted

The software takes an airflow measurement at power-up, and presumes this value is the “zero line” or baseline measurement for subsequent calculations. No quality check or reasonableness

test[19] is done on this measurement. Subsequent calculations are compared against this baseline measurement, and the difference is the change in airflow. If the airflow is slower than the baseline, this would result in a negative flow measurement, so the software simply adjusts the negative reading to a positive value.

If the measurement of a later baseline is taken, and the measurement is declared in error by the software, the software simply uses the last "good" baseline, and continues to read flow values from a declared erroneous measurement device.

Range Limits Are Substituted for Incorrect Average Measurements

In a manner similar to the diagnostics, voltage values are read and averaged into a value. If the resulting average is a value out of range, the averaged value is changed to the low or high limit value. If the value is out of range after averaging, this should indicate a serious problem, such as a failed A/D converter.

It is hard to imagine a calculated average occurring outside of the data input limits, since the data inputs are being forced within limits. Claiming "This cannot happen", means there should be no test for the condition in the code. If it does happen, then this substitution of values hides an obvious design problem.

[19] A type of test that determines if a value falls within a range considered normal or logical. It can be made on electronic signals to detect extraneous noise as well as on *data* to determine possible input errors.

Code Does Not Detect Data Variations

If the A/D Converter diagnostic is necessary because the readings in fact jump out of range, then it follows that other readings are within limits, but could vary widely. This condition is not checked, and calculations using these readings could be incorrect. (A test should be done to see if the reading is reasonable.)

If this "does not happen", then it follows that the diagnostic routines are not necessary because the converters are always valid measurements. Perhaps then the diagnostics are used as a sales feature to give false assurances to the customer.

Zero-Crossing Detection

The software uses data from one frequency cycle of the IR device at a time. It looks for a positive-going change in the readings, that is, the values must be negative, then zero or positive. This is the timing point for the start of the 2 Hz cycle.

The problem is the software takes a success-oriented logic path to the measurement. It does not measure if there were false readings, and it does not check the elapsed time of the cycle to the negative-going zero crossing, as well as the subsequent return to zero on the positive path again. The software presumes the wave form will always have the same characteristics.

Further, the clock interrupt mechanism is not synchronized with the described wave form characteristics. The clock interrupt is based on 8.192 ms and 64 interrupts are supposed to

measure the 2 Hz cycle. But the actual timing of the measurement is 1.907 seconds, almost a tenth of a second short to acquire the entire wave form.

Error Detection Logic

The software design detects measurement errors, but ignores these errors unless they occur a consecutive total number of times. For example, in the airflow measuring logic, if a flow measurement is above the prescribed maximum value, it is called an error, but this error must occur 32 consecutive times for the error to be handled and displayed. This means that the error could occur 31 times, then appear within range once, then appear 31 times, etc., and never be reported. The software uses different criteria values (e.g. 10 instead of 32) for the measurements of the various Alcotest components, but the error detection logic is the same as described.

Software Does not Insulate/Protect Modules or Data

A way to prevent unforeseen defects is to code safeguards as a standard part of the coding process. For instance, data required only by a single subroutine should be declared in the subroutine, so that other subroutines cannot inadvertently change it.

The Alcotest software relies heavily on "global" data variables, which means any of the functions can inadvertently modify a data value not part of that routine's process. If only the bare essential data were declared global, and the rest local storage, this significantly reduces the problem. Nevertheless, using any global data requires safeguards.

Code dealing with vectors and arrays and using pointers or indexes need to check that the calculated index or pointer really is within the array bounds. Arguments passed to subroutines should be analyzed for correctness. Likewise, values returned from functions.

Timing Problems

The design of the code is to run in timed units of 8.192 milliseconds, by means of an interrupt signal to a handler, which then signals the main program control that it can continue to the next segment. The interrupt goes off every 8.192 ms, not 8.192 ms from my latest request for a time delay.

The more often the code calls a single 8.192 ms interrupt, the more inaccurate the software timing can be, because the requests from the mainline software instructions are out of phase with the continuously operating timer interrupt routine.

Analogy: You have an "alarm clock" that has an hour hand, minute hand, and second hand, and the hands move in a stepping motion from one tic to another. This clock chimes every minute on the minute (when the second hand reaches 12). If you want this clock to time a minute and tell you when the minute happens, and the second hand is at 5 seconds when you submit the request, then the alarm will go off 55 seconds later, which is close enough. If instead, you request the minute timing and the second hand is at 55 seconds, then your "minute timer" in that instance would be only 5 seconds long. This means the analysis of the mainline code dependent on the interrupt timer operation will reveal that the actual timings will vary widely, and be inconsistent.

Common-sense practice is to write interrupt handler software as small units of code that operate quickly, and for a short period of time. The Alcotest external interrupt routines are very lengthy and are written in C instead of assembly language, which will increase the execution time and the memory used. There is a possibility that the interrupt handler can operate on an interrupt, and take too long and miss the next one. In fact, there is a test in the code to implement a "short rti", meaning return quicker from the interrupt, because this might be a problem in some circumstances. The short path is achieved by disabling large parts of the interrupt handler, which implies that an error could occur getting the handler back to the full function.

If this processor latches or saves missed time interrupts, all the interrupts will be processed, but the processing time will be stretched, and processes like A/D measurements could be affected. If this processor does not latch interrupts then some will be missed, and the time measurements will be shrunk, and device measurements will be affected accordingly. This depends on how the processor handles interrupts, and this specific information is not in the datasheet for the microprocessor, so that particular issue cannot be addressed precisely.

It has been claimed that the software checks devices 128 times a second, but the timing of the period is based on 8.192 ms, not 8 ms, and the software actually interrupts 122 times a second, a 5% difference between described and actual.

Other example timing sequences describe a 100 ms operation, but actually the timer value is 98.3 ms, approximately a 2% difference. (Further, in the example above describing out-of-phase

requests also happens, then the timer value could be off another 8.192 ms, to a value of approximately 90ms, which is 10% short.)

Uncalled Functions in Source Code

Fifty-one functions (subroutines or programs) of 475 total functions are not used and are still in the source code. This is 10.7% of the functions. These functions are compiled and taking up memory space in the device.

Code Disabled, not Deleted

Similar to the above, in this case functions or blocks of code have been disabled by comment markers, and the compiler does not produce instructions for the disabled code. However, because the code is not removed, this indicates uncertainty on the part of the developers. Either the code change is an experiment, which might have to be undone, or a capability is being temporarily removed for a customer or version that might have to be restored, or, this is the only way to document a now unused process.

Data Records

The data records (breath test results) for tested subjects are stored in random access memory (RAM) in the unit, and presumably, this is non-volatile RAM, meaning the data is retained in the machine when power is turned off. The data storage scheme is such that a corruption by incorrect storage or external factors such as static electricity, etc. can result in the loss of all retained data records.

The records are of unequal (varying) length, and the program determines the location of the next record by using information in the current record. A single corrupted record can cause the chain to be broken and lost.

It is possible with special case software to recover most of the data records, but that software process is not included in the Alcotest software.

It is not mentioned anywhere in the user or training manual that the unit can be turned off and not lose data.

Simultaneous Operations

The operation of the software is designed as "one-thing-at-a-time" processing. Even the interrupt or clocked operations delay the processing of the main line of software execution. This means that one device such as the fuel cell is measured sequentially, at a different time from the infrared sensor or the flow or pressure sensor. This implies that each test in the sequence has to go according to plan, or the timing may be thrown off for other tests. Certainly, it is important to note that many of the measurements that could be taken nearly simultaneously are not.

Allocation of Functions to Source Files is Unevenly Distributed

Seven source files contain 26.5% of the code, by lines of code. The rest is spread over 88 source files. This implies that these seven are large modules, with a lot of code that is confusing to read and difficult to maintain. Modern coding methodology allocates one executable function to one source file.

Global Variable Declaration

The Alcotest source code indicates use of an older method of declaring global variables in a C source file for programs (called global.c), and then using another header file (global.h) to make the list of global variables available. This is a problem if the two files are not updated at the same time, with the same characteristics.

A suggested method uses one source file, then uses a macro or #define statement to actually define memory space in one module, then outline the space for other subservient modules.

Defects In Three Out Of Five Lines Of Code

A universal tool in the open-source community, called Lint, was used to analyze the source code written in C. This program uncovers a range of problems from minor to serious problems that can halt or cripple the program operation. This Lint program has been used for many years. It uncovered that there are 3 error lines for every 5 lines of source code in C.

RF Interference Measurement

While RF interference is not a New Jersey option, the coding is a further representation of the quality of the software. This should also be considered if New Jersey acquires this feature in the future.

The code describes a test for measuring RF interference. According to the comments and notes in the source code, the test measures 10 readings, then averages the10 values. The program repeats this procedure a total of 16 times, then uses the 16 averages to determine that RF is present.

In fact, the code takes 160 readings, using every tenth reading and discarding the rest, instead of averaging values ten at a time. This method does not integrate 160 data readings and is not a valid measurement of RFI, according to the code design outlined in the comments.

The design criteria indicate that RF interference is only declared if present for a quarter-second or longer. If present for periods of less than a quarter-second, no RF interference is declared.

Source Matches Binary Image

Drager supplied a binary image that purports to be the file used for the Alcotest 7110 MK III NJ Version 3.11 software. While the binary image could not be compared and verified to an actual operating unit, the evidence supplied strongly indicates that the source code supplied ultimately produces an image that matches the image supplied, and the source code has several mentions of the New Jersey version. This exercise established that the source code supplied had a very high probability of being the NJ 3.11 version.

As further supporting data that neither internal (Drager developed) nor external (ISO 9000, etc.) development standards were followed, the compilation and linking instructions supplied had fatal errors; the instructions supplied were followed precisely, but would not compile. Also, the procedures would not run on a Windows 98 machine, but did work eventually on a Windows 95 machine.

SECTION V SUMMARY / CONCLUSIONS

1. As a matter of public safety, the Alcotest should be suspended from use until the software has been reviewed against an acceptable set of software development standards, and recoded and tested if necessary. An incorrect breath test could lead to accidents and possible loss of life, because the device might not detect a person who is under the influence, and that person would be allowed to drive. The possibility also exists that a person not under the influence could be wrongly accused and/or convicted.

2. The findings of this code review are self-evident. The code cannot be tested exhaustively and meticulously tested, because we run out of time. The other option is a review with a development standard. Using a development standard would leave "footprints" in the code, source directories would be read-only and be governed by a configuration management system, problem reports would be public knowledge, and there would be either an internal or external review board to review and recommend changes. (For instance, there is no evidence that version 3.11 came from a baseline of version 3.10. It could just as easily be based on version 3.8, missing changes from version 3.9 and 3.10. This does not suggest that basing versions on an earlier version is a bad idea rather that the documentation trail is incomplete.)

3. Since there is no ongoing development and review program, defects are probably "fixed" more than once, as there was no documentation to review and discover the problem occurred and/or "fixed" in the past. Furthermore, there is a high likelihood that when

bug fixes are introduced to the software or when new features are being supplied to a customer, new defects or bugs are also added.

4. There are no identifying notations or warnings that sections of the code should be considered proprietary and/or confidential. This means a new programmer would have no idea these lines should be kept from competitors, and could possibly disclose them. If a military standard were applied to the Alcotest source code, there would be strict guidelines applied to mark confidential, secret and top secret sections. In the commercial arena, confidential code is still marked inside the code, to provide some protection in case of court actions.

5. This machine has both a fuel cell and an IR detector. The microprocessor only processes a finite number of different types of electrical signals. There are digital inputs (0 or 5 volts), digital outputs (0 or 5 volts), interrupt signals (0 or 5 volts) and the A/D converters, which are continuous voltages from 0 or 5 volts. The input, output and interrupt lines are common to all microprocessors; no secrets there. The IR sensor, flow sensor, and the fuel cell output a value from 0 or 5 volts read by the A/D. Not only is this processing commonly understood, the A/D manufacturers will supply code snippets, support, and electrical diagrams for all to use.

6. The computing foundation devices for the Alcotest, the microprocessor, the A/D converters, and the RS232 serial devices are technologically obsolete, so we cannot understand how code for an obsolete chip and A/D converter would be used by a

competitor to gain a development and economic advantage. If we were to contemplate building a competing device, we would use none of this code and start from scratch. When Drager needs to replace the A/D and or the processor, there will be significant hardware and software changes.

APPENDICES

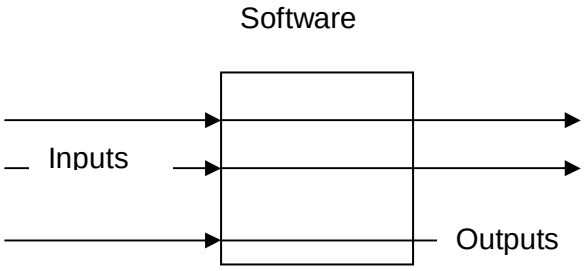
Appendix A: Standards & the Software Life-Cycle

Software Development vs. Electrical and Mechanical Design

With a mechanical design, either it works or it does not. Electrical designs involve devices with a finite number of connections and formulas, such that most of the problems can be eliminated at the schematic phase. However, software's appeal is that a computer chip can now have thousands or millions of ways to change the sequence of electrical inputs and outputs to the chip, because the memory of the computer chip can contain millions of individual instructions and data.

Single Inputs Generate Single Outputs

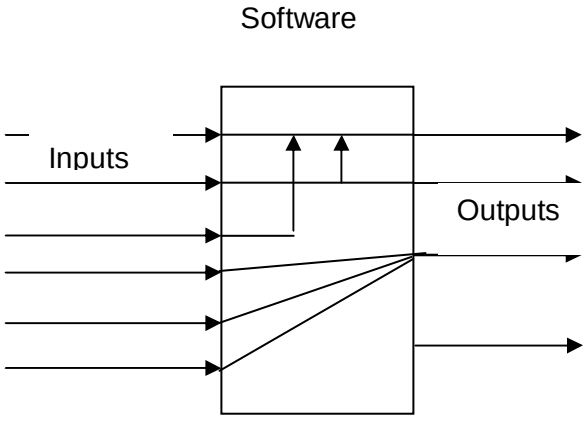
However, let us pretend we did not have a computer, and we looked at a simple device to implement.



In this diagram, the box represents our theoretical software, and the arrows represent three separate inputs, that generate three separate and unique outputs. So, if the inputs were light switches that were either on or off, then you would only have to test six switch positions, and observe the outputs, and the testing is done.

Multiple Inputs and Outputs with Multiple States

But in this example:



The requirements are more complicated. Note that some inputs continue to become unique outputs, but many of the inputs are tied to more than one output. Therefore, if one of the inputs were a temperature value, ranging from zero to one hundred, then there are at least a hundred values that have to be tested to see what the outputs should be. Further, if a test took ten

seconds, then testing the entire temperature range would take 1000 seconds, or over sixteen minutes—and you still have not tested the other inputs and outputs.

The important point is: A complex computer program cannot be thoroughly tested, because the testing time for each additional input and output increases exponentially and there simply is not enough time in the universe to complete the testing of one cycle of the software operation.

Standards and Testing

Faced with this scientific finding, software developers looked for methods to reduce systematic defects in software development. A conclusion reached was that requirements were not well defined, and requirements that were available were not understood completely by software engineers. Concentrating first on the requirements, then the design, and then the methods and processes evolved into a set of standards governing what is commonly called the Software Life-Cycle. This is simply defining phases of a software program's existence from creation through implementation, evolution, and finally phase-out or obsolescence.

Historical Design and Coding Methods

The earliest software development was done by trial-and error, where coding was implemented, and then changed when it did not work. The coding style then appears as though it were a newspaper article written by a columnist and then changed by 15 editors. It has a choppy sense to it, and it is difficult to maintain, and pass along to new programmers when the development programmer leaves the firm. It is also characterized by coding a few source files with very many lines of code, so the overview of the effort is lost in the details in the module.

The next stage was "structured programming", or the "top-down" method. This means you start with the highest or most abstract level, and then you add functions to the software as you see the need as you bore down into the details. This was an improvement in that requirements were better understood, but the design process did not continue down into the lower levels, and the programmers still programmed by trial and error.

Next was object-oriented programming. This method is listed here for a more complete history. Object-oriented design is a theoretical way to isolate programs, code, and data from each other so that fewer careless programming errors occur, and the code is more readable, maintainable, and, in theory portable. This means a program used on the Alcotest might be used on another device without modification, and the theory continues that testing would not be necessary. It has not lived up to this promise (programmers are not using previously developed code, code generated requires huge amounts of memory, and it runs slowly). Object-oriented code will never be used on a breathalyzer-type device at current prices.

Existing Software Standards in Industry and Government

Since complex software cannot be tested completely, by definition, standards have been developed that have been proven to improve the software development if they are strictly followed. Several organizations have approved the use of aircraft, medical, armament, and diagnostic devices if these standards are followed, even though it is understood that the software cannot be fully tested.

It is a common fallacy that applying standards, using a long requirements discovery and design phase, will make a software project late, and cost more, because people are sitting around dreaming up designs, and no coding is started, which then delays the start of testing and deployment.

Instead, it has been shown that employing these standards and development methods actually reduces the cost of a development project, because more requirements are discovered earlier, and then the coding matches more of the complete design, instead of discovering through the middle of a project that something was omitted, and now the design has to be revamped.

Therefore, an important reason to employ standards is to reduce cost on a project, as well as ensure adherence to design and requirements.

Here is a list of some widely used software development standards:

IEC 61508 Functional Safety International Standard

This is an internationally developed standard regarding safety of electrical devices and software.

ISO 9001 (International Standard for Requirements)

While it sounds like this standard only applies to the Requirements Phase, actually it applies to the entire life-cycle (see below). The idea is to check the coding phase, for instance, and verify that the requirements are still met.

IEC 62304 (FDA, but also International)

Standards regarding software in medical devices like insulin pumps.

DO-178B (FAA & International)

Mostly used on avionics and other processor controlled devices used on commercial aircraft, but the standards are also used for devices for private aircraft.

DOD-STD-2167 & MIL-STD-498

Software standards used by the U.S. Military and also some government law enforcement agencies.

NHTSA, NTSB and OIML

These agencies have not caught up to the current level of technology, and do not recognize software as a significant component. Instead, software is grouped with the overall device, like the box cover or main power supplies. These agencies do not have their own unique, developed, software development standard. Further, there is no requirement to use one of the known standards listed above. There are standards for tires, etc, but no standards for software development are listed or required, although a developer can of course volunteer a standard to use for the development.

.

Software Life-Cycle

No matter which of the above software standards is chosen, the standards revolve around the "Software Life-Cycle", which governs how software is designed, coded, tested, documented and maintained. The following is a brief description of the stages common to the standards cited above:

The segments here are a combination/hybrid of the Royce Model (waterfall) and the Boehm Model (spiral)

Requirements Analysis Phase

What does the software need to do? Do not pay attention to the coding needed, computer used, etc. Focus only on what needs to be supplied or computed, all in an abstract vein. List all inputs and outputs from the software under design.

Design Phase

Research and decide the overall software approach to the problem, and assign functions and tasks to individual software modules (programs). As each sub-function or sub-process is required, the design methodology is applied to the smaller and/or subservient software units.

Coding Phase

At this point, code (write) the computer instructions. Analysis methods and coding standards are also applied, to ensure that the code written meets the requirements and the architecture of the design phase.

Unit Testing Phase

This is testing of small pieces of the software. The unit tests are applied as the coding of the module completes, according to the plan. Since much of the software is missing at this point, temporary software and "stubs" are employed. It may not be possible to test all of the unit code until more modules become available. There is much simulation software used at this phase. Unforeseen errors arise, and this will require a return to the requirements and design phase. Eventually the same unit under test returns to this point, and tests are repeated to see if the code changes are effective.

Black-Box Testing Phase

At some point, nearly all of the planned software modules will be considered ready as a candidate for deployment. The software aggregate is tested with simulated inputs at the beginning of this phase, and eventually the aggregate will be tested in the production device with a simulated operational situation. Unforeseen errors appear, and we go back to Requirements, Design, and Unit testing again.

Black-Box testing means that software is tested as though the code is not known by the tester. The tests stimulate the software with inputs, such as pressing a keyboard key, and observing all the results from that input change. The reason the code is hidden is that knowledge of the code and its operation would unwittingly skew the designs of the tests run to have a higher probability

of success, and would not as easily discover unforeseen defects or defects from seemingly impossible conditions.

Deployment

The unit goes out the door to the customer. Installation instructions, user manuals, and other support items should be ready at this phase.

Defect Tracking and Configuration Management

Problems are discovered and reported from the customers and users. You need to record the defects and which software version they appear in. (The developer should have also been tracking errors in the earlier versions for internal use, and to make sure that "fixed" errors do not reappear. This is known as preventing a regression error.)

Defect Review and Implementation of Changes

Management, or a Change Review Board, in the case of some government agencies, review outstanding defects and fixes, and authorize the production of a new version. The process now starts over at the Requirements phase and proceeds forward from there.

Appendix B: Lint Program Analysis

Lint is a program designed to analyze software coded in C. It looks for inconsistencies in the coding, such as when a module passes a two-byte variable to a program expecting a one-byte variable.

An error detected by lint does not prove that the program will not execute, but it is a strong indication that the code is not reviewed and standards are not followed. This is the difference between a defect and an error.

A counter argument to the analysis of lint is that the program is very fussy, and only discovers minor problems. Using a real-world example to demonstrate the value of a lint analysis: If a Master Degree candidate's final thesis was filed such that one or two words were misspelled, then it would probably be overlooked and the information contained in the thesis would probably be relevant and accepted. However, if a thesis is filed with ten or more misspellings or grammatical errors or faulty references per page, then the author/candidate would at least appear careless, and the thesis would not be accepted by the evaluation committee without rigorous improvements. *There were 19,400 lint error messages on the ninety-five modules, a strong indication that even an internally developed standard by Drager personnel was not followed, and not much consideration was given to the possible defects in the code.*

In order to better describe the defects found by the lint program in the following sections, a brief description of software design is necessary. Computers use registers and electronic subsections to perform operations or instructions contained in the memory area of the computer, and these operations are further performed on memory locations containing data and (hopefully) not instructions. Examining a randomly chosen, consecutive four-byte area of memory reveals a random set of binary values. An examiner or tester would find it impossible to determine the intended use of the memory area examined just by looking at the binary locations. The area could be an integer variable, a floating-point number (a real number in mathematical terms), a series of four alphabetic characters, an array of small-value numbers, or an array of logical or binary values (true or false, yes or no, on or off). The computer can treat the same data as representations of very different data types. To reduce confusion, the C language allows the programmers to set aside sections of memory for a particular purpose and for a particular type of calculation or data representation. This is described as declaring variables belonging to a standard family, or type. Examples of variable types are: integers (int), character, (char), and real numbers (float, and double). It is considered safer to declare all the variables needed for an integer calculation in a formula as integer, and not use integers, characters, floats, etc. in the same calculation. There are times where it is necessary to convert one type of data to another, but functions should be coded to do this, so that the maintainer of the code understands that types are being converted. Formulas such as:

IntegerVariable = 'ABC' + 21 – 3.2;

should not be considered a normal way to do business, even though the computer does not consider this an erroneous statement, and will execute it.

A universal software design uses the concept of a "main" program or function, which transfers execution or calls a lower level program. This sub-function or subroutine then performs a non-trivial operation, and returns the result for the main program. Data values are passed between functions using argument variables. The purpose of these variables is to be the conduit of information between software functions.

Again, if the receiving ("called") subroutine expects one type of variable, such as character data, and if the sending routine sends an integer value, then the receiver gets a value that is the wrong number of bytes, and possibly is an invalid character value.

The following is a representative, partial list of the defects detected by the "lint" program, with a brief description of the possible impact:

Mismatched Function Argument Types

As described above, this means that the argument values passed to a sub-function do not match the declared arguments of the called function. There are also cases where the function returns a different type and value than the calling function expects.

Some compilers automatically convert the mismatched types, but this should not be considered a solution. Compilers often make very general assumptions about conversions. Further, this is a hidden operation by the compiler, so programmers returning to change a software module would wrongly assume that the arguments matched.

The preferred way of dealing with this is to code specific operations to convert one data type to another, or to rewrite the code to match the data types.

Local Functions Declared External

Functions or subroutines may be declared local or external (global). In the former, the function is to be used by one or two routines, and all would be declared in the same source module. Global functions are used by all the routines, implying that the module came from the software supplier as a utility routine or library function, or, the developer codes a routine that is intended for wide use in the program.

In this case, the function is intended to be locally used, but is declared an external type, such that any other routine could call it, perhaps with unintended results.

Local Variables Declared Global (External)

This is similar to the previous problem. In this case, the C language provides a mechanism to declare variables characterized as local or global in scope. Many designs use globally declared variables so that many routines can contribute to the same variable. One problem with this is that functions can make mistakes and address the wrong memory locations, causing changes to variables that were not intended.

Specifically, this message means that a variable used by only one routine has been declared as a global, which means any routine may alter it. If it had been declared local, then there is less of a chance of unintended alteration.

Formula Results are Mismatched Type for Target Variable

The C language tries to internally convert values so that the greatest amount of information is retained. When an integer value and a floating-point number are combined in a formula, for example, then the C compiler uses the floating-point type, because more info is retained. But if the result is intended for storage in an integer, then a conversion is needed again, and precision and accuracy may be lost in the conversion.

Formulas should take variable type into account, not just the mathematical and logical aspects of the formula.

Memory Leaks Detected

Some software design schemes take all of the unused memory in the computer, and make it part of a resource called dynamic storage. This allows programs to manipulate strings or arrays of data memory without needing to know the total size of the memory needed. A program then asks, for example, 100 bytes of memory for a display message. After the message is not needed, then the dynamic memory is "released" back into the system.

If a program does not release allocated memory, then eventually the computer loses track of the available memory, and the system cannot perform operations, because all memory is "lost". This is called a memory leak in the jargon. Lint has detected these memory leaks in the code.

Variables Assigned Different Types, Depending on Conditions

There are instances where variables are changed depending on which logical branch is followed. For instance, an integer might be assigned an integer value on one path, and a character variable on another branch (such as an if-then-else decision). The type of the value assigned should be consistent.

Arrays Initialized with Too Many Values

Consider a table with four rows and three columns. If the table is designed to contain certain values, then twelve values are required to fill the table. This lint message means that there are more values supplied than locations available to fill them like twelve or thirteen. It is not possible to tell whether the extraneous values are required and the size is wrong, or if values were accidentally typed in.

Tables are often used to calculate complex formulas, so this would affect the result. Also, it is possible that the extra values overwrite other memory locations and affect those calculations as well.

Comparisons are Different Types

This means that variables of one type, such as the integer 123, are compared against another type, like Boolean, such as one or zero (true or false), and the path of the program execution is dependent on the result. This is like comparing apples and oranges. The computer may decide the answer is fruit, but that was not the right answer.

Functions Passed Arguments that are Not Used

This implies that a function was originally designed to use a parameter or argument that is not needed. The developers decided to leave the argument in the source code. This implies that the developers could not find all instances of the function and change them to the new list of arguments, or felt the lack of change was benign.

The additional problem comes from a programmer new to the project that sees the unused argument in the subroutine, and now there is confusion. Was the missing argument needed somewhere else in the program? Can it be removed? Now the programmer has to search all the other source code to find a possibly missing variable.

Functions Called with Null or Missing Arguments

Similarly, there are functions that expect to receive argument variables, but the calling routine does not supply them. Instead a "null" or "void" value is passed, and the function tries to complete the task with missing values. If a function needs only three arguments in one phase of the program, but needs five variables in another phase, then two functions should be coded so that the argument lists match.

Table Index Variables could be Negative Values

Arrays are a straight-line collection of similar data. Arrays can be one-dimensional (a vector), two-dimensional (a table), or have n-dimensions. Array elements are located by the code: Array[row][column] for a table example. By definition, an array can have zero to n elements. This lint message says that negative values are used for selecting the element. This usually results in a data fetch from an undefined memory location, containing a random value.

Variables are Declared but not Used

This is another indication of unplanned, trial-and-error programming. During the development process, an idea requiring a variable might have been conceived. However, the code associated with the variable was never written or was discarded.

Table Bounds Checking

Since a calculation to access an element of a table could result in an invalid value, there should be code that checks the element index to ensure that the calculation lies within the table. The code should take corrective action if this is detected.

Dynamic Memory not Initialized

A variable type called a pointer is used to hold the address or memory location of any data type. Pointers are usually used to hold the actual address of the memory allocated, or to hold the

address of a data structure that contains multiple different data types, or they are used to hold the address of an array element, because it is more readable in the C language to use pointers.

This lint message means that a pointer is used, but it does not point to a valid memory address, and/or the memory allocated could contain random values and has not been set to a value consistent with the data type being used. For instance, character strings should be initialized to ' ' (blanks) before use.

Dynamic Memory Bounds Checking

Just as the array calculations should check that the indexes calculated lie within the dimensions of the array, a function should perform checks on dynamic memory.

If a function requests and gets twenty-five bytes of memory, then the code using that memory should always check the address in the associated pointer variable, to see if the pointer is within the twenty-five bytes requested.

Functions Return Values That Are Ignored

If a function performs work, but is not supposed to return a value, then it should be declared as returning a type "void", or, no return argument. Also, the called function should not have a "return" statement that passes back a value.

Comments are Nested within Comments

The point of this lint message is that there appears to be the start of a comment block, and then another start of a comment block appears without detecting the end of a comment block. A comment is an area of text in the source code that the compiler will ignore, and not translate into instructions. Comments are used to leave design notes or programming notes inside the source code.

This probably does not indicate that the code has serious problems in this area. However, a section of code might have been inadvertently commented out, which could have consequences. At the least, it indicates lack of coding standards.

Variables are Assigned with Loss of Precision

Similar to an example quoted above, this message means there is a definite loss in precision and accuracy when values are assigned to a variable, following a formula in the code.

Numeric or Character Data is Treated as Logical/Boolean

Boolean or logical data is a data type that has only zero and one as the legal value. The data is often interpreted as "on or off" or "true or false", etc. The point is that if a variable is Boolean, it can only have two meaningful values. There are Boolean-related formulas which employ operators like "or", "and", and "not".

This message means that a data type other than Boolean is used in a logical expression, for example:

b = "true" or "123";

instead of

b = “true” and not “true”;

As expected, logical operations are used to make decisions, and the first example value does not make sense.

Dynamic Storage not Defined

This is a different problem regarding initialized storage. Imagine a dynamic memory region, which contains a data record for a test subject. The first field in our example contains the name, which is stored in another dynamic memory area. The second field holds the address, again in a dynamic memory area, and so on through the rest of the data record.

The problem comes when the initial memory allocation is made. The compiler will allocate the data record, but will not allocate the data areas for the fields unless the memory allocation routine is specifically called.

Possible Use of Variable that is Not Initialized

This means a variable is declared, and initialized (supposedly) to a reasonable value. However, the compiler does not do this; it is the programmer’s responsibility. This usually affects program loops. Start at zero, do something five times, quit. In this example, there is no guarantee the counter starts at zero.

Functions Passed Undefined Dynamic Memory

This complicates the undefined dynamic memory problem cited above by passing the undefined data to a called function, which further complicates the problem.

Infinite Loops

This means that a programmer coded a loop of instructions from which there is no apparent escape. Sometimes the escape mechanism is from an interrupt or invisible operation; this mechanism should be noted in comments. There are also other ways to code this kind of logical construct.

Incorrect Comparisons

Example: There are tests or comparisons like: is this unsigned int (an integer that is only zero or a positive number, never negative) greater than or equal to zero? This implies that the programmer does not know the C language well, or he expected a negative value but did not know the compiler would convert to the unsigned int type.

Macro Errors

A Macro is a special set of codes operated on by the compilers preprocessor. This means the compiler looks over the code once, and does the instructions contained in the macros. These instructions typically code things like the version number, license, and conditional statements like: do this only for this state.

Lint has detected that macros in the Alcotest code would cause a compiler error in modern compilers. Apparently, the IAR compiler accepts it, or there is an option that ignores this error.

Possible Function Return Failure

There are functions that return without choosing the value that is supposed to be returned. This would possibly return undefined values, with unpredictable consequences.

Multiple Definitions of the Same Function or Variable

It is possible to declare both a local and a global version of the same variable, or the same function name. This causes confusion as to which variable or function is supposed to be referred to.

Functions Return Values Different from Declared Type

This means that a subroutine returned a value such as a character value, when the calling routine expects an integer.

Source File Contains Invalid Characters

This means that there are characters, usually unprintable, or, some characters belonging to the ASCII character set are not legal according to the compiler's parsing rules.

Overlapping Memory Areas used in Function or Formula

Consider a vector/array containing: A B C D E F G. If we wrote instructions to copy the "A B C D" into the locations starting at "D", then unpredictable results can occur, because you would copy A into where D is now, and then B into E, then C into F, but when you get to where D was, there is an A now. This is because the source locations "overlap" the target locations. So the final result would be A B C A B C A, instead of A B C A B C D.

Macros Used instead of "typedef"

Finally, this message says that a preprocessor statement is used to declare a special variable, instead of a typedef statement, which is better suited to the purpose. The difference is that the former is not processed by the compiler, but the latter is. This means that the compiler will detect the special type and generate conversions where necessary.

Appendix C: Lint Errors by Module

(Provided Under Separate Cover, Sealed Envelope)

APPENDIX D: GLOSSARY OF TERMS

GLOSSARY:

A/D (Analog to Digital) Converter: A device that processes a continuous signal (like AC signals), and outputs a discrete value that approximates the signal at the measurement point. The A/D Converters used in this device convert from an input range of zero to five volts to a value ranging from zero to 4095. Each step value in the output number represents 0.00122 volts. In addition to being limited to only 4095 values, the A/D cannot produce output values continuously. It needs time to convert the values. Therefore, the device really samples data at a regular period, instead of producing a continuous stream of values.

Assembler: A computer program that translates a text document into binary instructions. The text document is formatted to contain one computer instruction per line. This allows a programmer to be very precise in programming a computer, but the language is error-prone by nature.

Comment: Software languages use what are called "reserved words" to process into machine language. Examples of reserved words are: "for", "if", "else", and "while". The compiler assumes all input is either reserved words, or data variables (in general). The compiler allows special symbols that programmers use to tell the compiler: "Ignore this next section of input, I just want to write a note to remind myself what is being done here."

Compiler: A computer program that takes a text document (like a Notepad document) as input and produces intermediate low-level (assembler) instructions, which are then translated into binary codes that the microprocessor can execute.

Function: (Also subroutine) This is an executable unit or component of the software. Execution passes from the main line to (usually) many functions, one at a time. Conventions and protocols are used to "pass" values from the main program to the subroutines and back. Functions commonly perform mathematical operations, and return results, but functions can also perform logical or analytical functions, without returning a result.

Interrupt: An electrical signal that causes the microprocessor to save its current state, and jump to execute a special routine to handle an urgent request or process, and then return to resume normal execution. Often, programmers also refer to the interrupt handling routine as an interrupt, also.

Linker: The compiler and assembler produce components that can communicate with each other, but for technical reasons it is difficult to arrange the components in memory and keep track of each other. The Linker takes the components as input, and produces a file that is one long representation of the program, which can be loaded into computer memory, EPROM's, or on to a disk. The Linker is the final production step, and the output is executed on the computer.

Memory Banking: This concept is unique to embedded systems. In an Intel-type PC, the processor can store a great deal of data, currently in the gigabyte range. In a typical embedded

system, the processors can only process 65,536 locations. Most applications need more storage. Banking is a process where a memory chip is used that is larger than the processor's range is used to store data. This process acts like a "moving window", and presents regions of data to the processor one segment at a time. (Note: the processor also controls the bank selection, so selecting the wrong bank will select the wrong data.)

Module: (Also called a source file). This is a human readable file, which is processed by a compiler to produce executable code. A module may list one or more functions or subroutines.

Appendix E – Curriculum Vitae
John Wisniewski

Mr. John Wisniewski, with a B.A. in Computer Science from SUNY in Potsdam, NY (1976) is simultaneously a Software Engineer and an EPROM[20] programmer. In addition to possessing a solid background and impressive track record in EPROM coding, his expertise includes a background in assembly, C++ coding, along with the ability to customize existing software applications on firmware

Mr. Wisniewski has over 30 years of experience as a programmer and 15 years in product development/entrepreneurial projects. He has been an embedded and assembly language programmer for over 30 years now, and has planned, developed, and manufactured products using firmware and EPROM's.

An individual programming in EPROM and familiar with firmware is rare however an individual like Mr. Wisniewski, who has successfully developed products using firmware and EPROM's is almost non-existent. He is truly expert in his field.

The following represents a summary of his experience with

Applications
Systems Engineering
Software Engineering
Electronics Hardware Development
Testing and Troubleshooting
Reverse Engineering
Product Development
Database Applications

SUMMARY of APPLICATIONS:

Spacecraft Measurements, Data Collection and Telemetry
Commercial Aircraft Condition Monitoring Flight Recorders
Defensive Avionic Systems – Jamming, Countermeasures
Military Communications and Intelligence Systems
Secure Network-Based Mission Planning Systems
Medical Instruments for Surgery – Ultrasound Technology
Wireless Communications Controllers and Tactical Displays
Signal Processing Systems utilizing Fourier Transforms
Electro Mechanical Motion Control Loops and Systems
Interactive User Interfaces – Voice Recognition and Control
Internet-Based Monitoring and Control of Remote Machines
Client-Server Internet Transactions, Web Site Deployment
Fully Automated Telephone Systems – Custom Applications

[20] An **EPROM**, or *erasable programmable read-only memory*, is a type of computer memory chip that retains its data when its power supply is switched off. In other words, it is non-volatile.

SUMMARY of ACTIVITIES

Systems Engineering – Conceptual Design and Development
Software Engineering – Planning, Programming, Testing
Electronic Hardware Engineering – High Speed Logic, RF
Hardware / Software Integration, Testing and Documentation
Reverse Engineering – Systems, Software and Hardware
Manufacturing Engineering – Process Control, Vendor Interface
Selection and Procurement of Materials, Parts and Components
Program and Project Management, Design Team Leadership
Configuration Management, Version Control, Documentation
Product Approval and Certification: FCC, FAA, FDA, DoD

SYSTEMS ENGINEERING

Requirements Definition – Marketing Liaison
Conceptual Design – Project Engineering
Definition of Detailed Systems Specifications
Development of Systems Block Diagrams
Hardware / Software Interface Definitions
Preparation of System Software Specifications
Software Structure Charts & Data Flow Diagrams
Development of Comprehensive User Manuals
Adherence Monitoring to MIL-STD-2167A
Systems Design Incorporating MIL-STD-1759A
System Software Review for DO-178B Compliance
Preparation of System Software Certification Plans
Review and Analysis of Software Life Cycle Data
System Safety Assessments & Reliability Analyses
Hardware Range and Resolution Analysis
CPU Speed and Configuration Analysis
Analysis of Mass Data Storage Requirements
Capability Maturity Model (CMM) Assessment

SOFTWARE ENGINEERING

Real Time Executives – Embedded Controllers
Device Driver and Interrupt Handler Design
Handlers for Hard Disks, Modems and Tape Drives
Direct Memory Access (DMA) Processing
Programming of EPROMS, Flash Memories
Microprocessor Programming – Assembly, C, C+
Test and Diagnostic Software Development
Design of Graphical User Interfaces (GUI)

Data Acquisition and Control – Telemetry
Multiprocessor Configurations – Reliability
Optimization of Memory Structure for More Capacity
Multitasking and Interrupt Handlers for Special
Applications – such as Units Designed to
Run without an Operating System

ELECTRONIC HARDWARE DESIGN

Microprocessor Application Architectures
Digital Logic Design for Computer Interfaces
Sensor and Signal Conditioning Circuit Design
Analog Circuits for Receivers and Transmitters

REVERSE ENGINEERING (examples)

Disassembly of Operating System Codes for Driver Change
Reverse Engineering of Military Communication Systems
Regeneration of Missing Systems Documentation
Porting of Software Code from one Language to Another

TESTING and TROUBLESHOOTING (examples)

Usage of Electronic Test Equipment for Problem Tracking
Conceptual and Detail Design of Custom Test Equipment
Computer Configuration for Testing and Monitoring

PRODUCT DEVELOPMENT (examples)

Small Remote Controlled Electro-Mechanical Robots
Network-Based Remote Machine Monitoring and Control
Voice-Operated Remote Controls for Electronic Equipment

DATABASE APPLICATIONS (examples):

Inventory Forecasting Systems for Reorder Planning
Automated Billing for Material and Services Provided
Identification Card Information Processing and Evaluation
Telephone System Time Charge Calculation and Allocation
Secure Access Control and Documentation Systems

HARDWARE and SOFTWARE USED

Microprocessors: ……..	Atmel Tiny12, etc., Dallas 80C320 Intel 80x86 Family, 8080 Family Intel 8051 and Derivatives Motorola 68360, 68xxx Family, 6805, 68HC11 Texas Instruments TI9900, 6502 Zilog Z80 Family
Mainframes: ……...	Data General, VAX, IBM 360/370, Modcomp
Software Languages: ……	Assembly for Processors Listed, PL/1, FORTRAN, BASIC / Visual Basic, C, C++, Java, Perl, HTML, Command Line Language for Several Operating Systems
Development Tools: ……..	Microsoft Visual Studio, Microsoft Visual Basic, Microsoft Visual C++, Microsoft Front Page, BRIEF Editor, Watfor C+IDE, Netbeans IDE for Java Development, Solaris Visual, Solaris Configuration Management Tools, Microsoft Source Safe, Beyond Compare Difference Tool, C-DOC Documentation Generator, Perl Scripts for Source Code Generation, Flash Development Suites
Operating Systems: ……..	MSDOS, VRTX, Macintosh pre-OSX Windows 3.11 / 95 / 98 / NT / 2000 / XP, UNIX, Linux, Solaris, VAX, Modcomp,
Bus / Interfaces: ……	ISA, RS232 / RS 485, GPIP, 1553 Bus, PC Parallel Port, VME
Laboratory Tools: ……..	Multimeters, Spectrum Analyzers, Logic Analyzers, Oscilloscopes, Protocol Analyzers, etc.
Databases: …….	SQL Server, SQL Query Language, Access, FoxPro, Paradox, NDBM

EMPLOYERS, CLIENTS, PAST AFFILIATIONS and PROFESSIONAL ORGANIZATIONS

AIL Division of Eaton Corporation, Deer Park, New York

AIL Division of Eaton Corporation, Edwards AFB, California

Association for Computing Machinery, New York, New York

Cardkey Systems, Simi Valley, California

COMCO, Inc., Burbank, California

Coto Interpreting, Glendale, California

MDA Technologies, Camarillo, California

MDA Technologies, Woodbridge, Virginia

Medical Technical Products, Irvine, California

NASA / Jet Propulsion Laboratory, Barstow, California

NASA / Jet Propulsion Laboratory, Canberra, Australia

NASA / Jet Propulsion Laboratory, Pasadena, California

Northrop Grumman, Dumfries, Virginia

Northrop Grumman, Woodland Hills, California

Teledyne Controls, Copenhagen, Denmark

Teledyne Controls, West Los Angeles, California

US Marine Corps, Camp Pendleton, California

US Marine Corps, Point Mugu Naval Air Station, California

Voice Control Products, Inc., Monterey, California

Voice Powered Technology, Canoga Park, California

John Wisniewski
Resume

Security Clearance:
Secret—Active

Education
B.A. Computer Science, SUNY Potsdam, NY 1976

General

Will take on challenging work others avoid. Winc Research has a broad base of experience, with a deep understanding of the individual fields. Winc Research is always interested in new areas of study. Winc Research is committed to thorough, detailed, exacting standards of project development.

Skill Set

Assembler/Binary/Firmware/Core Dumps	C++ and C, Java, Basic, FORTRAN, PL-1
Training & Instruction	SQL, SQL Server, Access and general DB
Technical Writing	Sound processing and recording
Machine Tools (Mill, Lathe, Manufacturing)	Military, Commercial, Scientific and Entrepreneurial Environments
Oscilloscope, emulator, logic analyzer, spectrum analyzer, PROM programmers	Reverse Engineering of code and systems
Configuration management, problem tracking and problem resolution	Electrical Assembly
Intel, Motorola, Phillips, Modcomp, Atmel processors	

Client, Work and Project History

LaundriMate (Winc Research 1996-present) laundry monitoring product receiving three US patents. Winc Research did all of the hardware design, searched for suitable voices, recorded the phrases and word fragments in three languages, and developed the sensor system for the appliances. Development of this device/product included obtaining FCC approval for Part 68 (Winc produces its own modems) and Part 15 residential emission standards. Winc Research is the only manufacturer, and over one thousand units have been shipped and installed in the United States (over 30 states) and Canada since January 1998. Once installed, no units have ever been returned for repair. A major upgrade in 2005 allowed for Internet and e-mail monitoring of the machines, in combination with telephone voice prompts. For a demo, call 310-391-7306, or on the Internet, http://www.laundryalert.com, then enter the password "stan9568".

Commander's Tactical Terminal (CTT) and Joint Tactical Terminal (JTT) (US Marine Corps and Contractors 1998-present) are two related radio receivers that process military intelligence from satellite broadcasts. The software controls and configures the receivers, and then processes the binary information into text-readable messages on the fly. The software then forwards the data over an Ethernet network to other computers for analysis. This software provides filtering of data, and multiple format conversion simultaneously.

MedScale (Winc Research 2002-present) is a medical monitoring device for Congestive Heart Failure patients. The device is in the beta-test stage, with four prototypes completed. The device will report daily patient weights as a first indicator of condition changes in congestive heart failure patients. Using technology leveraged from LaundriMate, the device will input data into a remote server and provide analysis tools for cardiologists. Winc Research is developing the server software, including the telephony receiving the equipment, and developing the case, hardware and embedded software for the medical device.

Infinity Cataract Surgery machine (Medical Technical Products 2000–present) is a device using ultrasound to remove the lens of the eye for treatment for cataracts. Winc Research designed and implemented the SVGA graphics elements (sprites) with TFT touch screen and mouse interface, operating in a MS-DOS-like environment, as well as all of the functional and operational software for the device.

SS-2 Surveillance system (US Marine Corps and California Microwave 2000-2002) is a small portable embedded NT Server system for radio signal interception (intelligence surveillance work), based on PC-104 boards. Winc Research provided software, electrical assembly and testing/verification services.

Advanced Mission Planning Software (US Marine Corps, Army Special Forces, California Microwave 1998-2002) is a system to allow better mission planning and communication over an Intranet, using either the Netscape or Internet Explorer browsers. While the system was primarily coded in HTML, it needed Common Gateway Interface (CGI) modules coded in C++ and using ODBC database interfaces, to maintain security. Winc Research designed, coded tested and delivered this portion of the system, as well as tested the HTML scripting.

SCSM system (US Marine Corps, California Microwave 1998-2002) Reverse-engineered SIGINT Common Sensor Module (SCSM) software from Digital Access Corporation, a collection of DCOM servers and clients to perform signal intelligence communication and acquisition. After the reverse engineering phase, Winc Research developed an interface module to perform communication between California Microwave's Common Gateway System and the existing SCSM systems.

KCAL Television Membership Card system (KCAL-TV, 1996-2000) Designed and implemented a turnkey database system for KCAL television, with automated telephone data entry. This database program and supporting software supported 8 telephone lines and 1000 calls per day, and Winc Research processed over 320,000 individual calls. Also designed, manufactured and implemented prize machines for the KCAL Card program for use at promotional events. This system tracks membership attendance and awards prizes for each promotion run. This program is Visual C++ MFC based and uses database, audio, and video multimedia applications.

Interpreter Assignment System (Coto Interpreting, 1998-1999) is a TSAPI-compliant Computer Telephony program to answer calls and track billing for an on-line language interpreter service. This program answered calls from clients, looked up the client information in an Access database using the Caller ID, and looked up and called appropriate interpreters for conferencing, then timed the calls for client billing. The program was Windows-MFC and ODBC based, with database object support.

Print Buffer Computer (AlphaMerics, Inc. 1997) Designed and implemented 80x86 based C and C++ programs for PC-104 based systems for Alpha Merics, Inc., including an HPGL plotter and printer-buffering program to support Alpha Merics' plotting tables. Winc Research also developed a calibration package to measure the flatness and linearity of the plotting surface before system shipment.

S-300 Badge/Card system (CardKey, 1997) is a Z-80 based embedded access-control system to implement security and attendance monitoring for businesses, coded in C and assembler. The system supported 30,000 badges (cards), and implemented remote monitoring through telephone lines. Designed and implemented several components of system including report printing, remote access through modems, RS232 monitoring, RS-485 protocols, and password encoding.

HUMS system (Teledyne Controls 1993-1996) is a VME-bus-based embedded system for the US Navy CH-46 helicopter. This was a 68360 based processor to monitor performance of a neural network based vibration and regime analysis system developed by the Navy, and also monitor performance of a Teledyne Health and Usage Monitoring System (HUMS) Avionics system. This monitoring program was used to

evaluate these two systems in a fly-off type competition. The project was coded primarily in C and machine language, including several IBM PC based RS-232 simulation routines for development testing.

Winc Research also had to code the embedded software for the TI-9900 8-bit processor for HUMS system for Teledyne Controls, including data acquisition of sensor data, and recording data onto flight recorder media. This system was coded entirely in machine/assembly language.

Micro-Sandblasting (COMCO, 1996) Designed and developed several robotic process-control systems for COMCO, Inc., a maker of micro-abrasive systems. These systems are in use at Sandvik, a maker of lathe and tools, and at Caterpillar's gasoline engine plant in Rockford, Ill. In addition to coding and testing, Winc Research provided on-site troubleshooting and support at the Caterpillar plant.

VCR programmer; Business reminder (Voice Powered Technology, 1993) Implemented voice recognition technology for Voice-Powered Technology, including a business reminder now on the market, and a voice command based VCR controller, which has been discontinued.

Various Toys and Products (Voice Control Products 1994-1996) Implemented voice recognition technology into several toys and consumer products for Voice Control Products Incorporated, of Monterey, CA. Various 8-bit microprocessors were programmed, including 68xx and 8048. All coded at assembly or machine language level.

Programmable DFDAU (Teledyne Controls 1990-1991) Developed customized routines for Digital Flight Data Acquisition Unit (DFDAU), an avionics unit to record FAA required data to the flight crash recorder. This unit records general data information to the additional data recorders used for airline maintenance and analysis. Programs were coded in C, and developed under VAX VMS.

Designed and implemented a customer programmable version of the DFDAU software, which reduced the amount of coding necessary to implement common changes for airline customers. Also designed and developed a C++ Object Oriented ground system to modify the programmable software.

US Navy F-14 (Teledyne Controls 1990-1991) Worked on several military projects for Teledyne Controls, including the US Navy F-14 monitoring system, which is based on the 1553 bus, VRTX, and coded in 8080 assembly language and C, and display software for the USAF C-17 transport, written in 68000 assembly language and C.

SOCS (Telos Corp. 1989-1990) is orbital modeling and prediction software to support RCA data communication satellites. This system, in FORTRAN, helped maintain geo-synchronous orbit position. Knowledge of physics, orbital mechanics and time calculations including Julian dates, was essential for software analysis and modification.

Radio Science Ground System (JPL, Telos Corp 1981-1990) Cognizant Design Engineer at the Jet Propulsion Laboratory (JPL) for the Deep Space Network Radio Science data acquisition software during the Voyager encounters at Uranus and Neptune. The computer-controlled subsystem was coded in HAL-S, a NASA-unique block-oriented programming language, and Modcomp assembly language. Design and implementation supported Radio Science experiments at Venus and Mars during this period, including support for USSR science experiments. Position included travel to Australia to train technicians and troubleshoot before encounters.

Telemetry Ground System (JPL, Telos Corp 1981-1990) Worked on subsystem software for the DSN for the Mark IV upgrade project. System coded in Assembly language.

Monitor and Control System (JPL, Telos Corp 1981-1990) was the major architectural addition for the DSN for the Mark IV upgrade project. Before this system was developed, operators had to manually configure up to 12 computers for a single track of a probe such as Voyager. Played key roles in system design, coding, test and qualification, and developing coding and documentation standards for development group. Coded in HAL-S and Modcomp assembly language.

USAF B-1 bomber countermeasures system (AIL division of Eaton Corporation 1976-1981) Designed, coded, tested, documented, and supported embedded system for AN/ALQ-161 defensive avionics system for USAF B-1 bomber. This was a defensive radar jamming system for the aircraft, coded in assembly language.

Designed, developed and supported a data acquisition system for the flight test program of the AN/ALQ-161 system to evaluate system performance, coded in FORTRAN.

Appendix F
Base One Technologies
Credentials

Base One Technologies, founded in 1994, is a 13 year veteran at providing a broad range of information technology services including application engineering, systems engineering, Independent Verification & Validation (IV&V) and consulting services. Over the past 13 years, Base One consistently stands out due to our knowledge, expertise and our ability to locate and secure top talent. We work on the critical systems necessary to keep our customers on the cutting edge of technologies and maintain connectivity. Base One furnishes leading edge technology and services to the Financial/Banking Industry (Merrill Lynch and Citibank); and the Telecommunications market space (MCI/WorldCom (now Verizon) and AT&T).

We currently consult with and provide services to the Federal Government for the Department of Defense (DoD), US Army Corps of Engineers (USACE); Federal Civilian agencies - Federal Aviation Administration (FAA), the Department of Homeland Security (DHS), and the Federal Trade Commission (FTC).

We are a "right-sized" company with the bench strength to meet most client needs and the ability to expand or contract staff to maintain strong business acumen and provide for an equitable Return On Investment. When customers come to us they know we will provide the right person for the tasking. If that person is not currently within Base One's employ, we recruit them. In order to provide for unique skill sets, we maintain on-staff a team of technical recruiters who routinely provide us with top level consultants for highly specific needs.

Our experience in the specific area of IV&V work includes

- Project Plan Review
- Requirements Analysis
- Requirements Tracing
- Milestone Reviews
- Architecture Design Analysis
- Software Design Analysis
- Software Code Review
- Metrics Development
- Metrics Measurement and Assessment
- Test Witnessing
- Test Planning, Execution and Reporting
- Training and Documentation Evaluation
- Site Acceptance Testing
- Defect Investigation
- IV&V Laboratory Support

Our specific work in the area of IV&V is synopsized below.

- **Citibank SSA Electronic Benefits Transactions (EBT)**
  - Comprehensive Platform Evaluation
  - Certification of Compliance with Security Requirements
  - Compliance Assurance to Social Security Administration standards
  - Integration and Component Level Testing

- **DoD PenRen: Communication Command Survivability Project:**

- Evaluation of IT products and services to meet DOD standards
- Ensure performance to design, cost, schedule and performance specifications
- Security Certification and Accreditation for IT systems
- Projects assessments, process and performance audits

•MCI/WORLDCOM - Network Consolidation, Migration and Redesign
- System Architecture, Software Design, Integrated Products Analysis
- Network Connection Compliance
- Life Cycle Management Analysis
- Vulnerability Assessment

•DHS: Homeland Secure Data Network (HSDN)
- Conduct Security Test and Evaluation
- IT Information Assurance conducting Certification and Accreditation (C & A)
- Ensure policies, procedures, physical mechanisms are established and operating properly
- Perform risk analysis on sites and systems, develop a risk mitigation plan
- Specifications include government regulations and guidance: DITSCAP, NIACAP

•Army Corp of Engineers: Operations Division Portal
- Established Verification and Validation plan
- Authored Test Specification (with Agreed Requirements)
- Executed Functional Tests
- test plans and cases
- test activities and processes
- Test Logs, Incident Reports, Summary Reports

Corporate Clients

Merrill Lynch	Citibank
Pfizer	SBC
Salomon Smith Barney	IBM
Harris Corporation	MCI/WorldCom(now Verizon)
USWeb	AT&T
JP Morgan	Guardian
Prodigy	SAIC
Sony	Northrop Grumman
GE	Quest Diagnostics

Government Clients

Department of Homeland Security
Department of Energy
US Army Corps of Engineers
Department of Defense: Pentagon
Department of Transportation: Federal Aviation Administration
Federal Trade Commission