

Light Water Reactor Sustainability Program

Preliminary Results of a Bounded Exhaustive Testing Study for Software in Embedded Digital Devices in Nuclear Power Applications



September 2019

U.S. Department of Energy

Office of Nuclear Energy

DISCLAIMER

This information was prepared as an account of work sponsored by an agency of the U.S. Government. Neither the U.S. Government nor any agency thereof, nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness, of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the U.S. Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the U.S. Government or any agency thereof.

Preliminary Results of a Bounded Exhaustive Testing Study for Software in Embedded Digital Devices in Nuclear Power Applications

**Carl Elks, Athira Jayakumar, Aidan Collins, Richard Hite, Tasio Karles, Christopher Deloglos,
Brandon Simmons, Dr. Ashraf Tantawy and Smitha Gautham
Virginia Commonwealth University
Department of ECE
601 W. Main Street
Richmond Virginia
PI: Dr. Carl Elks**

**In collaboration with National Institute of Standards Technology
Computer Security Division
Richard Kuhn, Diego Bouvier Burgueno, Raghu N Kacker and Jeff Voas
100 Bureau Drive
Gaithersburg, MD 20899**

September 2019

**Prepared for
Ken Thomas
Idaho National Laboratory
U.S. Department of Energy
Office of Nuclear Energy**

ABSTRACT

Under the Department of Energy’s Light Water Reactor Sustainability Program, within the Plant Modernization research pathway, the Digital I&C Qualification Project is identifying new methods that would be beneficial in qualifying digital I&C systems and devices for safety-related usage. One such method that would be useful in qualifying field components such as sensors and actuators is the concept of testability. The Nuclear Regulatory Commission (NRC) considers testability to be one of two design attributes sufficient to eliminate consideration of software-based or software logic-based common cause failure (the other being diversity). The NRC defines acceptable “testability” as follows:

Testability – A system is sufficiently simple such that every possible combination of inputs and every possible sequence of device states are tested and all outputs are verified for every case (100% tested). [NUREG 0800, Chapter 7, Branch Technical Position (BTP) 7-19]

This qualification method has never proven to be practical in view of the very large number of combinations of inputs and sequences of device states for a typical I&C device. However, many of these combinations are not unique in the sense that they represent the same state space or in that they represent state space that would not affect the critical design basis functions of the device. Therefore, the state space of interest might possibly be reduced to a manageable dimension through such analysis.

This project focuses on a representative I&C device similar in design, function, and complexity to the types of devices that would likely be deployed in nuclear power plants as digital or software-based sensors and actuators (e.g., Smart Sensors).

This report describes the preliminary findings of a study to support Bounded Exhaustive Testing using Combinatorial Test (CT) methods in addressing NRC regulatory guidance on software common cause failure. The report describes the process workflow, testbed architecture, tools, resources, and computing used to conduct an automated testing process for this purpose. The report further describes how the testing of a smart sensor (pressure transmitter) was conducted. It presents the test results, which indicate that this methodology has strong potential to support digital qualification with respect to software common cause failure assessment.

CONTENTS

ABSTRACT.....	ii
FIGURES.....	vi
TABLES	vii
ACRONYMS.....	viii
1. INTRODUCTION.....	1
1.1 Purpose.....	1
1.2 Test Objectives and Scope	2
1.3 Embedded Real time Systems and Testing.....	2
2. BACKGROUND: COMBINATORIAL TESTING.....	3
3. CONCEPTUAL STUDY PROCESS	5
3.1.1 Preliminary Concepts.....	6
3.1.2 Number of tests	7
4. TESTBED ARCHITECTURE	8
4.1 Test Workflow	10
4.2 Software Functions Tested.....	12
4.3 TESSY and ACTS tools.....	14
4.4 Test Interface.....	15
4.5 Processes Critical to Bounded Exhaustive Testing.....	16
4.5.1 Input models and Classification Tree Method	16
4.5.2 Test Oracle – from an automation perspective	17
4.5.3 Thread Level Integration Testing.....	18
4.5.4 Test Case Coverage.....	19
4.5.5 ACTS2TESSY Translator.....	21
5. Preliminary Results	22
5.1 Overview of the Systematic Testing Approach.....	22
5.1.1 Summary of Results.....	24
6. Details of the Preliminary Results.....	27
6.1 Issues Identified in Software by Testing.....	27
6.2 Unit Testing and Results	28
6.2.1 Function ms5611_kalman_filter: 2-way combinatorial testing	28
6.2.2 Failures with Infinity test result	32
6.2.3 Failures with ‘NaN’ (Not a Number) test result.....	33
6.2.4 Coverage Statistics.....	33
6.2.5 Function ms5611_kalman_filter 3-5 way combinatorial testing.....	33
6.2.6 Function ms5611_get_current_pressure 2-6way combinatorial testing.....	34
6.3 Component/Integration Testing and Results: ms5611_thread	34

7.	Other Findings	36
7.1	Next steps	39
8.	REFERENCES	40
	APPENDIX A	42
	EXAMPLE TEST REPORTS	42
	APPENDIX B – VCU Smart Sensor	1
1.	Introduction	1
1.1	General Matter	1
1.2	User Documentation	1
1.3	VCU Smart Sensor Components.....	2
1.3.1	Hardware Architecture	2
1.3.2	Software Stack Model.....	4
1.3.3	Real-Time Operating System – ChibiOS	4
1.4	Smart Sensor Threads	5
1.4.1	Data Flow	5
1.4.2	Communications Interfaces.....	6
1.4.3	User Data Logging Interface.....	6
1.4.4	Debug and Test Port Interface.....	7

FIGURES

Figure 1 Cumulative proportion of faults for t (number of parameters) = 1 . . . 6 [7].	4
Figure 2 Conceptual view of a bounded exhaustive testing process.	6
Figure 3 Baseline testbed architecture	9
Figure 4 MS5611 thread structure	12
Figure 5 shows the basic flow of the ms5611_thread functions. Items in red indicate low level functions which have not been tested. Green items indication functions that are the target for testing.	13
Figure 6 Breakpoint Feature - A failed test case can be debugged by using the breakpoint feature of TESSY [15]	15
Figure 7 On Chip Debugger Interface to Smart Sensor [15]	16
Figure 8 Variable value ranges as defined in the ms5611 device datasheet [17].	17
Figure 9 Classification tree for ms5611_get_current_pressure function	17
Figure 10 Epilogue code under each test case for ms5611_get_current_pressure function	18
Figure 11 Decision Coverage [19]	19
Figure 12 MC/DC Coverage [19]	20
Figure 13 Test coverage criteria selection	20
Figure 14 Test coverage with ms5611_get_current pressure combinatorial testing	21
Figure 15 Test coverage shown in control flow graph of ms5611_get_current_pressure function	21
Figure 16 Testing strategy	23
Figure 17 Test Interfaces for ms5611_kalman_filter function	29
Figure 18 Classification tree for ms5611_kalman_filter function	29
Figure 19 Snapshot of Component Test results of ms5611_thread	35
Figure 20 Sequence based testing for ms5611_thread in TESSY	36

TABLES

Table 1 Relationship between v and t for covering array tests	7
Table 2 Invoking conditions for function calls in the call graph	19
Table 3 Number of t-way Tests generated from ACTS tool for the functions in ms5611_thread.....	22
Table 4 Combinatorial Testing Experimental Study.....	24
Table 5 Summary of Results.....	25
Table 6 Software issues identified by testing.....	28
Table 7 Sample input values assigned to each class in the Classification tree for ms5611_kalman_filter function.....	30
Table 8 2-way Combinatorial test results for ms5611_kalman_filter function	31
Table 9 Coverage Statistics for 2-way Combinatorial testing on ms5611_kalman_filter function	33

ACRONYMS

API	Application Program Interface
BVA	Boundary Value Analysis
CA	Covering Array
CCF	Common-Cause Failure
CCM	Combinational Coverage Measurement
CFG	Control Flow Graph
CPU	Central Processing Unit
CSV	Comma-Separated Values
CT	Combinatorial Test
CTM	Combinatorial Tree Method
DUT	Device Under Test
EDD	Embedded Digital Device
EPC	Entry Point Coverage
FC	Function Coverage
HW	Hardware
I&C	Instrumentation and Control
I/O	Input/Output
JML	Java Modeling Language
JTAG	Joint Testing Action Group
MC/DC	Modified Condition/Decision Coverage
MCC	Multiple Condition Coverage
NASA	National Aeronautics and Space Administration
NIST	Nation Institute of Standards and Technology
NPP	Nuclear Power Plant
NRC	Nuclear Regulatory Commission
OCD	On Chip Debug
PIL	Processor in the Loop
RTOS	Real-Time Operating System
SCCF	Software Common Cause Failure
SUT	Software Under Test
SW	Software
SWD	Serial Wire Debug
UART	Universal Asynchronous Receiver/Transmitter

UAV	Unmanned Aerial Vehicle
USB	Universal Serial Bus
VCU	Virginia Commonwealth University
XLSX	Microsoft Excel File
XML	eXtensible Markup Language

PRELIMINARY RESULTS OF A BOUNDED EXHAUSTIVE TESTING STUDY FOR SOFTWARE IN EMBEDDED DIGITAL DEVICES IN NUCLEAR POWER APPLICATIONS

1. INTRODUCTION

As digital upgrades to U.S. nuclear power plants (NPPs) have increased, concerns related to potential software common cause failures (CCF) and potential unknown failure modes in these systems has come to the forefront. The U.S. Nuclear Regulatory Commission (NRC) identifies two design methods that are acceptable for eliminating CCF concerns: (1) diversity or (2) testability (specifically, 100% testability as identified in BTP 7-19, Rev. 7, “Guidance for Evaluation of D3 in Digital Computer-Based Instrumentation and Control Systems,” U.S. NRC, August 2016 - Accession No. ML16019A344). As pointed out in Reference [1], there is near universal consensus among computer scientists, practitioners, and software test engineers that exhaustive testing for modestly complex devices or software is infeasible [2], [3] – this is due to the enormous number of test vectors (e.g., all pairs of state and inputs) needed to effectively approach 100% coverage [4]. For this reason, diversity and defense-in-depth architectural methods for computer-based I&C systems have become the norm in the nuclear industry for addressing vulnerabilities associated with common-cause failures [5]. However, the disadvantages to large scale diversity and defense-in-depth methods for architecting highly dependable systems are well known – significant implementation costs, increased system complexity, increased plant integration complexity, and very high validation costs. Without development of cost-effective qualification methods to satisfy regulatory requirements and address the potential for CCF vulnerability associated with I&C digital devices, the nuclear power industry may not be able to realize the benefits of digital or computer-based technology achieved by other industries. However, even if the correctness of the software has been proven mathematically via analyses and was developed using a quality development process, no software system can be regarded as dependable if it has not been extensively tested. The issues for the nuclear industry at large are; (1) what types of software (SW) testing provide very strong “coverage” of the state space, and (2) can these methods be effective in establishing credible evidence of software CCF reduction. In the previous report, we identified several promising testing approaches that purport to provide strong “coverage”. Namely, among the methods reviewed were Combinatorial Testing (CT) methods that can achieve “bounded” exhaustive testing under certain conditions [6]. Additionally, the term coverage requires some farther elaboration as to classify among the several definitions used in the SW testing community – and we provide that discussion in latter sections.

In this document, we describe a well-formed approach for carrying out bounded exhaustive t-way testing to support an empirical study to collect data on the efficacy of CT methods for accomplishing bounded-exhaustive testing. In addition, we provide preliminary results and findings on the research to date.

1.1 Purpose

In this document we are focused on describing the realization of a testbed for conducting automated empirical software testing of embedded digital devices with respect to bounded exhaustive testing. We are mainly focused on methods that support or claim high levels of “coverage” approaching exhaustive testing or bounded exhaustive testing. By bounded exhaustive testing we mean:

Definition: The term *bounded-exhaustive* is used in relation to *software* testing. Software testing is considered *bounded exhaustive* when well-formed relations between input space and state space allow the testable state space to be reduced – enabling a feasible testable set. The bounded aspect relates to the lower bound of contraction on space sets using a set of well-formed inference rules. Typical methods used (among others) to achieve state space reduction include boundary value analysis, covering arrays, and equivalence partitioning. The key assumption is that the state space reduction process must preserve the properties of and among the elements from the original state space [7]–[9].

Definition: *Coverage* refers to the extent to which a given verification activity has satisfied its objectives. Coverage measures can be applied to any verification activity, although they are most frequently applied to testing activities. Coverage is a measure, not a method or a test. As a measure, coverage is usually expressed as the percentage of an activity that is accomplished, state space exercised or represented [1], [10].

1.2 Test Objectives and Scope

The approaches, methods, and technologies described herein are mainly focused on testing actual software for embedded digital devices—i.e., testing with actual inputs stimulating the software under test. Our objective for this research is to develop a test approach and methodology to enable a study on the efficacy of t-way combinatorial testing for embedded digital devices. To support this specification, we will develop questions to be tested by the study. These questions will be asserted in terms of statements that can be supported or refuted by the study.

Question 1: Can t-way combinatorial testing provide evidence that is congruent with exhaustive testing for an embedded digital device?

Under what assumptions and conditions for this claim to be true?

Question 2: Is t-way combinatorial testing effective at discovering logical and execution-based flaws in nuclear power software-based devices (device under test [DUT])?

The overall scope of this test specification is focused on t way combinatorial testing methods, technology and supporting tools required to effectively carry out a well-formed study to answer Questions 1 and 2. The scope of the research in this report is confined to bounded exhaustive (or also called pseudo exhaustive) testing on a subset software threads of the VCU smart sensor.

1.3 Embedded Real time Systems and Testing

Embedded systems and embedded digital devices differ from general purpose desktop computing platforms in several notable ways that challenge testing. First, they are often resource constrained in memory, processing, inputs and outputs to accommodate specific and specialized functionality. Secondly, the software structures are organized to do specific tasks in relation to their coupling to the physical world. Lastly, embedded systems are often real time or in-time computation sensitive, owing to the fact that they are controlling or monitoring physical processes. In total, these characteristics differentiate embedded digital devices from computing perspective from general purpose or numerical computation, explicitly embedded digital devices or systems follow what is called *the reactive computing model*. A *reactive system* is characterized by its ongoing interaction with its environment, continuously accepting requests from the environment and continuously producing results [11]. In reactive systems, correctness or safeness of the reactive system is related to its behavior over time as interacts with its environment.

Unlike, functional computations which compute a value upon termination, reactive programs usually do not terminate. If they do terminate, it is most often due to the fact that an exception event has occurred. Example applications of reactive systems include process control systems, monitoring systems, reactor protection systems, real-time systems, and telecommunication protocols. A reactive system has *state* which describes the current and past conditions of the plant (the physical system that is being controlled or monitored), (2) *environmental stimulus* (sensor measurements) cause *transitions* from one state to another state in the control or processing algorithm, and (3) transitions cause *reactions* (output commands) that dictate proper specified behavior to the plant.

The physical system or plant is an integral part of the design and the software of reactive system. Embedded digital devices are often real time meaning that there timeliness of responses to stimulus. Responses that are correct in value but late or early in time can cause the plant to deviate in expected behavior. Actions and reactions happen concurrently and over time, and the properties of time are an essential part of the behavior of the real time embedded system.

In real time software, the temporal aspects of physical interactions are dealt with by replacing time with ordering of actions and events. For example, in languages such as C, C++, and Java, the order of actions is defined by the program, but not their timing. This ordering abstraction is then overlaid with an program execution model which schedules threads or processes at specific discrete time instances, typically provided by the real time operating system or an executive. Together, both of these abstractions can create complex interactions and orderings that are difficult to comprehend from the designer and user perspective. Testing these interactions and orderings in a systematic way increases the confidence in the correct system functionality. Accordingly, testing methods that are consistent to the nature of real time embedded system interactions are required.

2. BACKGROUND: COMBINATORIAL TESTING

As real time software grows in size and complexity, testing all the interactions between the data, environment, and configuration is challenging. The studies conducted by the National Institute of Standards and Technology (NIST) [7] on software failures in 15 years of Food and Drug Administration Medical Device recall data concludes that a majority of the software failures are due to interaction faults arising from the interaction of a few parameters, mostly by two and three. According to the National Aeronautics and Space Administration (NASA)-distributed database, 67% of failures are triggered by a single parameter, 93% by 2-way interaction and 98% by 3-way interaction. Several other applications studied also depicted similar results, shown in *Figure 1*. Applying the rule that the interaction between t or fewer variables subsumes all of the interaction failures in software, testing all the t -way combinations of the variables can lead to “pseudo-exhaustive” testing of software. The combinatorial method, which involves selecting test cases that cover the different t -tuple combinations of input parameters, can lead to generating compact test sets that can be executed in considerably less time, while at the same time providing significant testability of the certain types of failures in software. Such failures are known as *interaction failures*, because they are only exposed when two or more input values interact to cause the program to reach an incorrect result. The key insight underlying t -way combinatorial testing is that not every parameter contributes to every failure and most failures are triggered by a single parameter value or interactions between a relatively small numbers of parameters.

On the basis of experimental data collected by NIST on a variety of software applications, as shown in *Figure 1*, it has been deduced that the cumulative percent of faults triggered in a software asymptotically approaches 100% when the number of parameters involved in the faults reaches 6. This, in turn, suggests that testing software with all possible 6-tuple input parameter combinations can lead to significant reduction of defects in the software.

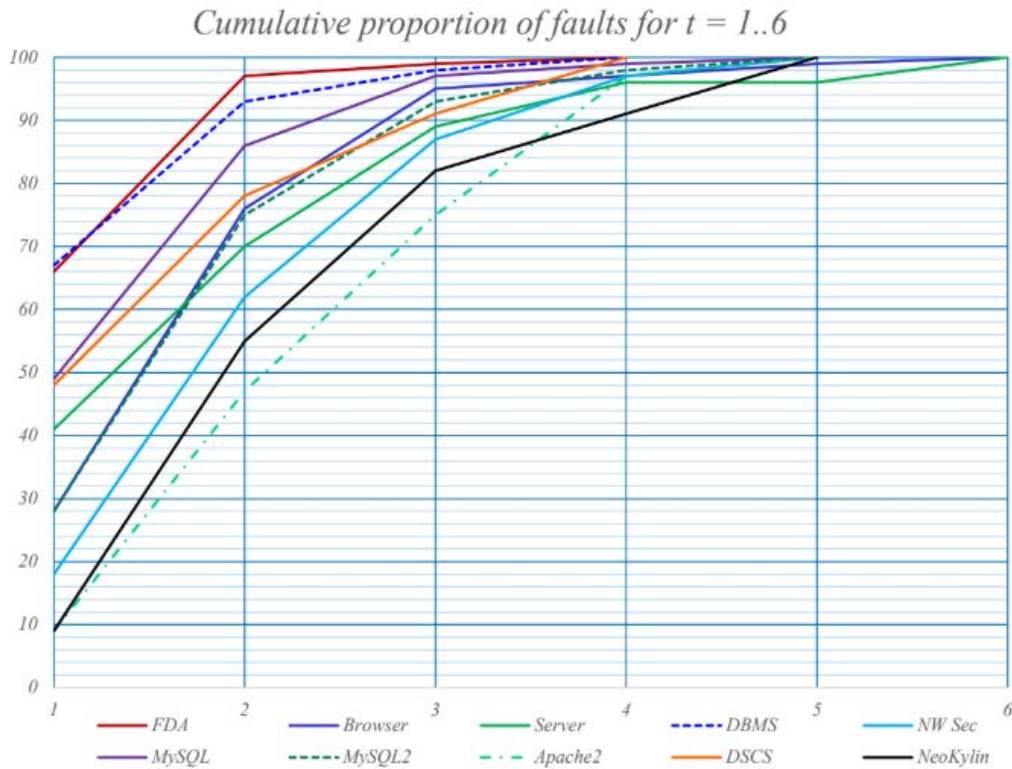


Figure 1 Cumulative proportion of faults for t (number of parameters) = 1 . . . 6 [7].

The two most-used combination arrays for combinatorial test-set generation are covering arrays and orthogonal arrays. Covering arrays $CA(N, t, k)$ are arrays of N test cases, which have all the t -tuple combinations of the k parameters covered at least a given number of times (which is usually 1). Orthogonal arrays $OA(N; t, k)$ are covering arrays with a constraint that all the t -tuple combinations of the k parameters should be covered the same number of times. The major elements of a combinatorial test model are parameters, values, interactions, and constraints [12].

The first step for creating a test model is to identify all relevant parameters. This should include the user- and environment-interface parameters and the configuration parameters. The second step is to determine values for these parameters. Using the entire set of values for all parameters would lead to unmanageable test suites and testing. Hence, to confine the values of the parameters to a necessary and tractable set, we need to apply the various value-partitioning techniques: equivalence partitioning, boundary-value analysis, category partitioning, and domain testing. As a third step, interactions between the parameters must be analyzed in order to generate an efficient set of test cases. Defining the valid parameter interactions and their strengths in the test model can aid in avoiding test cases involving interactions between parameters that actually never interact in the software and also in prioritizing test cases for closely interacting parameters. Specifying constraints on the interactions, which define the set of impossible parameter interactions, is also vital for obtaining the expected software coverage [12].

R. Kuhn and V. Okun's work on "Pseudo-Exhaustive Testing for Software" [7] discusses the concept of integrating combinatorial methods with model checking and presents the results of applying this technique on an experimental system. Model checking can be used for automatic test-case generation. The requirement to be tested is identified, and a temporal logic formula is described in such a way that the requirement is not satisfied. This formulation of the negative requirement will be the test criterion and will cause the software model to fail, thus causing the model checker to generate counterexamples that

can be used as test cases. By using t-way coverage of the variables as the test criterion, we can derive the combinatorial test cases. Temporal logic expressions in the form $AG (v1 \& v2 \& \dots \& vt \rightarrow AX \neg(R))$ which direct that for the input variable combination $(v1, v2, \dots, vt)$, the condition R should be false in the next step, has to be fed as input to the model-checker tools. Thus, the model checker will generate counterexamples that cover all variable combinations that satisfy R. The experiment conducted by Kuhn et al in using a Symbolic model checker to create pairwise to 6-way combinatorial testcases for a traffic-collision-avoidance system gives supporting results. It shows a 100% error-detection rate with 6-way combinatorial coverage of inputs. Although there were more counterexamples generated by the model checker than the actual t-way combinations needed, the number of redundant testcases were found to reduce as the input interaction coverage (t) increases.

R. Kuhn's (of NIST) and J. Higdon's research work on extending the application of combinatorial testing to event-driven systems, described in the paper, "Combinatorial Methods for Event Sequence Testing," [13], also proves to be noteworthy for systems of the type found in NPPs. Some faults in the software are activated only when there is a particular sequence of events happening, a very relevant condition related to NPP operations. Sequence-covering arrays can be used to test all the t-way order of t events in a software. The basic concept of sequence covering is that, if we have a two-way event testing, there should be a test case with $x \dots y$ such that y event occurs after x event. And there should also be a reverse order of the event occurrence $y \dots x$ where x occurs after y. Testing the forward and reverse order of occurrences for all the events with respect to all other events can help in detecting most event-driven failures in the software. The research paper provides mathematical proof that the number of tests only grows logarithmically with respect to the number of events. In summary, combinatorial, sequence-based testing has shown to be effective method for testing event sequence-based issues in real time software, thereby improving the coverage and efficiency of testing.

There has also been a lot of research in the field of studying and developing various algorithms for covering array-test suite generation, which include greedy algorithms and heuristic methods. Bryce et al.'s greedy algorithm for test-case generation in [14], which takes user inputs on the priorities of the interactions to be covered and which also allows for seeding of fixed testcases into the test set, is identified as another important work in the field of combinatorial testing.

3. CONCEPTUAL STUDY PROCESS

Figure 2 shows conceptually the study process we propose to address the objectives of the testing. The first step is to define all of the relevant parameters required for the test objectives. In this case the critical parameters are t, v, and n. Each of the variable space parameters (v and n) defines the input model. The input model is pre-analyzed (e.g., BVA) to determine equivalence partitions. The input model define is used to define the "list" of experiments – and this can be done parametrically (one factor at a time), or by Design of Experiment methods. The list of "covering" test vectors is generated to produce a reduced optimal set of test vectors. One-way experiments can be designed by varying the t variable for a given set of "experiments" – increasing t incrementally. The same can be done with the v parameter. These experimental test vectors are applied to the DUT. For each experiment executed, the DUT must start from a known good state. This usually requires the experiment automation instrumentation to issue reset before each experiment. Once the DUT is operational, the test vectors are applied. The outcomes of the DUT are observed by the test oracle or by assertions (maybe code based). A test oracle is a function or system that evaluates the outcomes of the testing results. The oracle makes decisions on pass/fail for a given test case, collects data for statistics, etc. The outcomes belong to three sets: Set of pass/fail, coverage metric (% of covering array), and a metric related to % of state space examined. The process continues until there are no more variations on the parameter sets OR the computational complexity exceeds the processing power to carry out the experiments. Data is post processed from the outcome space to determine if the experiments yielded evidence to support (or refute) the claims (test objectives). Prior to the experiments

being conducted it is often good practice to execute tests on baseline fault sets to determine the capacity of the method to detect faults. The key point is that the testbed implementation must be designed from experimental viewpoint with respect to the test objectives. Below we discuss our proposed testbed environment and process to support the execution of experiments.

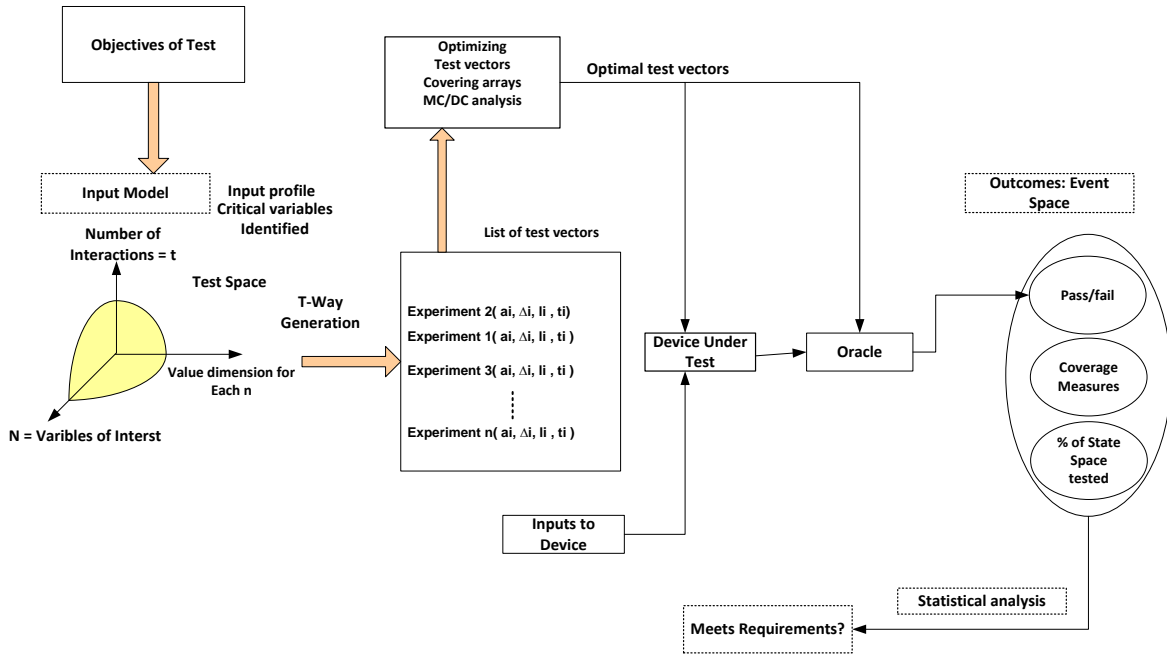


Figure 2 Conceptual view of a bounded exhaustive testing process.

3.1.1 Preliminary Concepts

To fully develop the idea behind this study, we first describe some essential material related to state space and interaction t-way combinatorial testing. Efficient generation of test suites to cover all t -way combinations is a difficult mathematical problem (NP hard: non-deterministic polynomial-time computationally complexity.). Additionally, contemporary software in most embedded digital devices is a combination of data types representing continuous variables (fixed point, floats), integers, Booleans which have possible values in a very large range. For effective reduction to a testable state space, the range of these values must be mapped to a much smaller range – possibly a few values. This is usually done though equivalence partitioning and sampling methods – another non-trivial problem. Most evident of all is the problem of determining the correct result that should be expected from the system under test for each set of test inputs. This is the oracle problem, how to determine when something is correct. Fortunately, most of these challenges have been addressed to the point where practical methods and tools supporting t-way combinatorial testing allow credible reduction of the input and state space. Nonetheless, like all testing methods, the creation of test oracles for combinatorial t-way testing is ongoing research activity.

In general, the number of t-way combinatorial tests that will be required is proportional to $v^t \log n$, for n parameters with v possible values each. The key parameter in these equations is v and t . Keeping v and t small reduces the “parameter state space.” t is a function of the logical behavior of the software. v is a

function of the data type space in terms of range of the data type. Normally, we would create partitions for each v that is minimally sufficient for testing. Example, if we had variable whose range was -10 to +10, then we might create a partition with the set $\{-10, -1, 0, 1, +10\}$ – 5 representative values. In this case, we have the min/max values, values close to 0, and 0. If we wanted to exhaustively test this range, then we need the full span of values (21). The issue in the design of this experiment is that we cannot use the full span of variables with a large range for comparative exhaustive testing. We have to find another way. One idea is to look at how the variable is used in the decision logic of the program. If the variable is a part of a condition or guard expression – then selecting a range of values on the condition and on either side of the condition might be sufficient for testing interactions. This is called *boundary value analysis* (BVA) to select test values at each boundary and at the smallest possible unit on either side of the boundary. The intuition, backed by empirical research, is that errors are more likely at boundary conditions because errors in programming may be made at these points. Additionally, we can expand the boundary analysis partition to include more representative elements – this becomes the basis for comparing to a “exhaustive set”. We could assert, the “pseudo-exhaustive” partition is defensible because every important element of the set is represented at least once, and the smallest units are used at the boundaries.

3.1.2 Number of tests

From [14], the goal to find covering arrays is to find the smallest possible array that covers all configurations of t variables. If every new test generated covered all previously uncovered combinations, then the number of tests needed would be:

$$\frac{v^t \binom{n}{t}}{\binom{n}{t}} = v^t$$

Since this is not generally possible, the covering array will be significantly larger than v^t but still a reasonable number for testing. It can be shown that the number of tests in a t -way covering array will be proportional to

$$\text{number of tests} \triangleq v^t \log n$$

Where v is the value span of the input variables or parameter (n), and n is the number of inputs parameters, and t is the number of interactions between parameters.

It should be noted that the number of tests grows exponentially with the interaction strength t , but logarithmic with the number of input parameters (n). The span of v determines the base of the value, which can have a growth effect of the number tests. Table 1 provides an indication on the relationship between v and t and the number of tests. We ignore n , since its contribution is logarithmic. Although the number of tests required for combinatorial testing can be very large (as illustrated below), with advanced distributed processing clusters and mapping software (like Gridunit or Hadoop) it is not out of reach.

Table 1 Relationship between v and t for covering array tests

$v \downarrow t \rightarrow$	2	3	4	5	6
2	4	8	16	32	64
6	36	64	256	1024	4096
10	100	1000	10,000	100,000	1,000,000
12	144	1728	20736	248832	2,985,984
16	256	4096	65536	1048576	16,777,216

For illustrative purposes suppose we have the following subset of variables taken from the VCU smart sensor:

- 20 Boolean variables - each variable takes on (T, F)
- 10 continuous time variables (float) – By BVA each variable is represented by 10 values.
- 10 Integer variables - By BVA each variable is represented by 8 values.

What would be the expected number of tests for a 4-way covering array?

- *BOOL number of tests* $\triangleq v^t \log n = 2^4 \text{Log}40 = 26$
- *INT number of tests* $\triangleq v^t \log n = 8^4 \text{Log}40 = 6562$
- *FLOAT number of tests* $\triangleq v^t \log n = 10^4 \text{Log}40 = 33220$

Total = 39,808 tests

Percentage of tests with respect to the separate equivalence partitions and t-way interactions

$$\begin{aligned} \frac{\text{Number of } 4\text{-way tests}}{\text{Number of } N\text{-way tests}} &\cong \frac{v^t \log(n)}{v^{n_{bool}}} + \frac{v^t \log n}{v^{n_{int}}} + \frac{v^t \log n}{v^{n_{float}}} \\ &\cong \frac{2^4 \log 40}{2^{20}} + \frac{8^4 \log 40}{8^{10}} + \frac{10^4 \log 40}{10^{10}} \cong .000019\% \end{aligned}$$

This low state space coverage result is interpreted as follows. If the equivalence and BVA partitions are well-formed for the program, and the covering arrays generate tests that cover all combinations - then only a very small percentage of well-formed test vectors are needed to perform as well as brute force all combinations testing – the essence of bounded exhaustive testing. This is the power of the test. The key assumptions are that reduction methods like BVA and equivalence partitions are well-formed for the program, and 4-way interactions are sufficient. In the case where 4-way interactions is found not to be sufficient, then performing t+1 (5) interactions is required.

Our approach to measuring the coverage and completeness of our testing is to leverage established combinatorial testing metrics such as variable value combination coverage, t-way combination coverage, and other metrics [NIST special publication 800-142].

4. TESTBED ARCHITECTURE

In this section, we present an implementation perspective of how to realize an automated test environment to support t-way testing on a real time embedded digital device. Our test requirements can be characterized by the following:

1. Test methodology is based on established science for safety critical systems and devices.
2. All testing methods are applied directly to the embedded digital device in real time.
3. High degree of observability and controllability of real time execution behavior of the embedded digital device.
4. Efficiency of testing is accommodated with advanced test automation methods.

5. A well-formed input model (testing profile) of the device for nominal operating, off-nominal, and out of range data ranges.
6. A well-formed test specification derived from the requirements and the source code.
7. Development of representative test values from established testing methods such as BVA, classification trees, and coverage metrics (e.g. MC/DC)

Figure 3 presents the basic test environment architecture with respect to tools, systems and components needed to support the BET experiment process. There are three “SW tools” that figure prominently in our testbed; (1) Razorcat’s TESSY, (2) NIST ACTS, and (3) Keil Interactive Debugger. The TESSY automated testing tool is developed by Razorcat [15]. TESSY’s primary purpose is for testing safety critical embedded system software. It provides a number of features to assist in automation, unit testing, integration testing, and regression. TESSY supports test management, including requirements, coverage measurement, and traceability. For our purposes, the automation features of TESSY are of significant interest. The NIST ACTS tools provide a full set of features and capabilities to support T-way combinatorial testing. This includes support for BVA, efficient T-way test vector, sequences, fault location and covering array generation. Finally, the Keil interactive debugger tool allows real time interfacing to the target device through the OCD ports of the device (e.g., JTAG, SWD). The “debugger” interface allows; (1) test vectors to be directly interfaced to DUT, (2) unobtrusive extraction and monitoring of all types of data including I/O data, internal variables, intermediate variables, state variables, conditionals and guards.

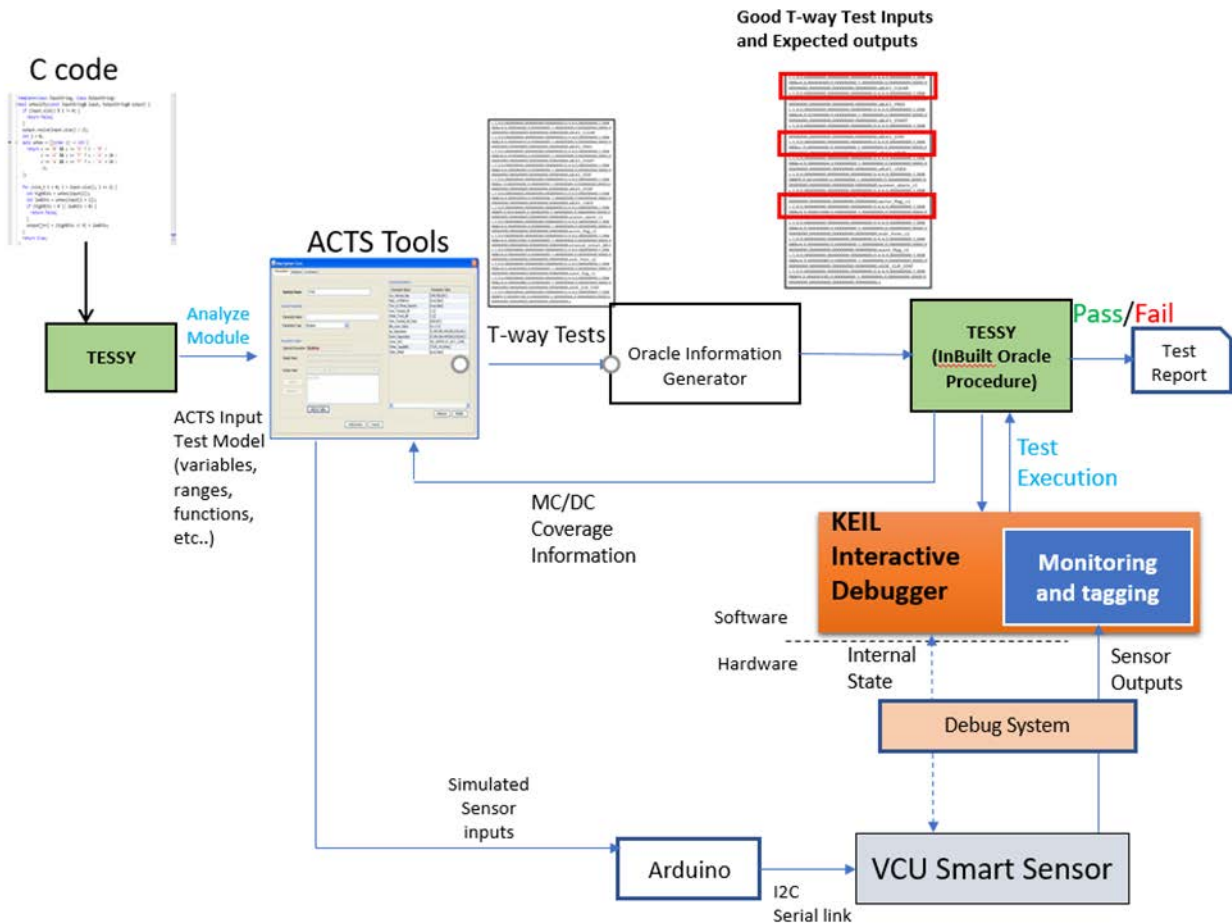


Figure 3 Baseline testbed architecture

Partially or fully automated testing schemes are required to optimize testing efficiency and carry out the large numbers of tests. Automation in this context refers at least to the following major items:

- Automatic generation of test cases
- Recording of Smart-Sensor state/output variables on a per-test-case cycle
- Appending test-case value with time-stamped smart-sensor data
- Execution of test cases
- Automatic comparison of Smart-Sensor results to Oracle

4.1 Test Workflow

This section describes the basic workflow of the automated test bed shown in figure 3.

1. **Development of the Input model.** The first step is to develop a input model for the testing process. This is partially facilitated with the TESSY tool and ACTS tools. We start with identifying the c source functions to be subjected to unit testing; the corresponding source files are loaded into the TESSY tool. TESSY tool analyzes the source files and populates all the local functions, external functions, external variables, global variables and macros. TESSY figures out the interfaces, input variables for stimulating the function and output variables for monitoring the correctness of the function. Input-Output model for each function in c file can be exported from the Test Interface Editor in TESSY. This export can be into any of the following formats: tcx, xls, xlsx, script, csv, txt or yml.
2. **Boundary Value Analysis (BVA).** Second step is to determine the representative values for the input model – this is the v parameter. Using the entire set of values would lead to infeasible test suites and testing. Hence to confine the values of the parameters to a necessary and tractable set, we need to apply the various value partitioning techniques like equivalence partitioning, boundary value analysis, category partitioning and domain testing. In TESSY, the input variable is assigned a sample value from the corresponding equivalent partition or a boundary value in each class. The classification tree information is exported from TESSY into an xml file and excel file. The xml file exported from TESSY that contains all the test data values is parsed by a script and translated into an input format understandable by the ACTS tool.
3. **TESSY-ACTS interface.** If using the TESSY tool for BVA on the input model, we must export the input model to ACTS. At present we convert the input model to format that ACTS Tools understands using a script file.
4. **Generate T-way Test vectors (ACTS).** Given the input model, ACTS will generate sets of t-way test vectors. The user decides on specific experimental parameters to control (e.g., unit test, t, n, and sequence). The xml file exported from TESSY contains all the test data values and is parsed by a script and then translated into an input format understandable by the ACTS tool. ACTS generates t-way test vectors that contains all t-way combinations of input values that are specified in the classification tree in TESSY. These t-way testcases are exported from ACTS and translated into TESSY import format by an automation script. A second pass will factor in MC/DC coverage information to refine test vectors.
5. **Sequence based Testing.** Sequence based testing is a time-based, test-sequence driven testing process. It concentrates on ensuring that with the right stimulus sequence, right reactions (state change/output change) happen at the right time in the right order. Combinatorial testing may not be applicable for functions like the thread handlers. Functions that have less of control flow logic, and

mathematical operations and instead have numerous external function calls, needs to be tested differently. In order to verify that the external function calls happen at the right conditions and in the right order, scenario-based test cases are built in TESSY

6. **Test Case import.** The complete test cases are then imported into TESSY in one of the following formats: tcx, xls, xlsx, script, csv, txt or yml.
7. **Oracles.** Even with efficient tools and procedures to produce input models, covering arrays, test sequences, the oracle problem is critical – testing requires both test data and results that should be expected for each data input. Much care should be given early and often on the “whats and hows” of the oracle – that is define what you want the oracle to do, and how it’s going to do it. In “*Automated Software Test Implementation Guide for Managers and Practitioners, STAT COE-Report-05-2018*” - appendix E defines a number of approaches for developing oracles for SW testing.
8. **Oracle Integration.** Once an Oracle approach has been selected, TESSY has basic features built in to support oracles in an automated testing framework. At the basic level, TESSY allows one to submit already generated t-way test vectors into an Oracle Information Generator which analyzes all test vectors and adds in the expect output values for all the test input vectors. This can be automated using some scripts or worst case can be done manually in TESSY itself (probably not feasible for large test sets. need to figure out). Important note: TESSY provides “hooks” to include oracles in the automated testing process. The design of the oracle is left up to the testers to realize.
9. **Test case execution.** Executing the tests requires an automated test environment where test vectors are submitted to the smart sensor, and results are cataloged. The key aspect is to collect data in manner that is tractable and supports the test objectives. TESSY runs test cases directly on the actual Smart sensor target (STM32F4 uC) by interfacing with the Keil debugger.
<http://www2.keil.com/mdk5/debug>. TESSY in concert with the Keil debugger identifies and time stamps all data. TESSY reads in the variable information within the smart sensor code via the Keil debugger interface.
10. **Outcome evaluation.** Expected output values are calculated automatically based on the oracle algorithm integrated into TESSY. The testcases are then run directly on the STM32F4 microcontroller using the Keil μ vision debugger software. TESSY compares the expected output values calculated using the algorithm and the actual output values and produces Pass/Fail results.
11. **Refining Test cases.** After the initial test execution, MC/DC coverage data can be generated via TESSY. This coverage data can be fed back into the ACTS tools (if possible) or analyzed manually to refine tests in TESSY to improve the test coverage on the code. Also, Automatic Test generator and Scenario based Test method in TESSY can also be used to improve coverage. Scenario Test cases in TESSY can be used to cover for scenario-based test cases that do not involve input value combinations but instead involve sequential invocation of various functions in a particular order. ACTS also has a similar feature to generate sequence based CT tests vectors.
12. **Analysis of results.** After all test cases have been executed then post analysis can proceed to compute various metrics on the efficacy of the testing. Since a comparative analysis of t-way combinatorial testing to exhaustive testing is desirable, we need to have as many t-way interactions as possible. Selection and analysis of metrics at the beginning of the experiment is important to ensure that experiment can support calculation of the metrics at post analysis phase.

4.2 Software Functions Tested

For this phase of the research we applied our testing strategy to one of the three main threads of the VCU smart sensor – MS5611 thread. Two main functions were tested: `get_current_pressure` and `ms5611_kalman_filter`. Figure 4 shows the call graph structure of the thread, and Figure 5 shows the logical structure of the thread.

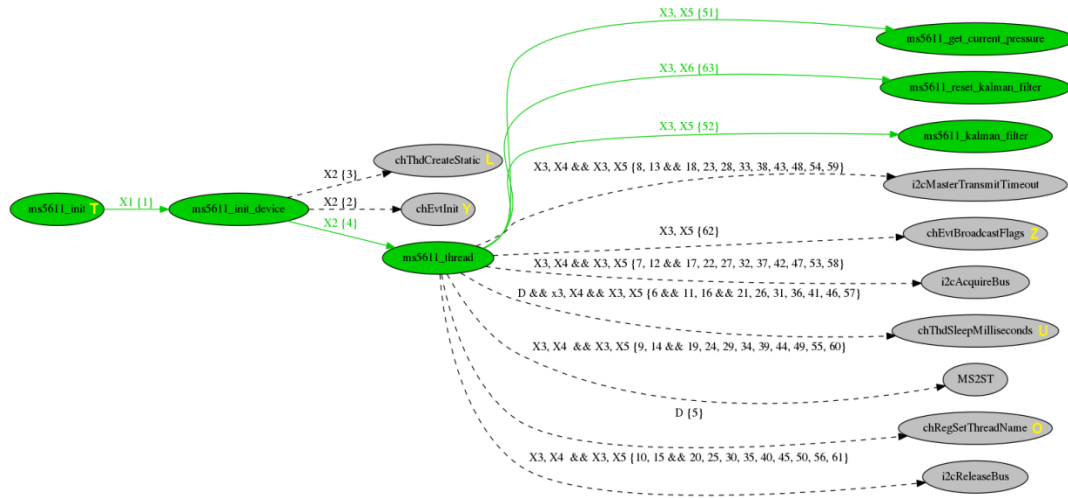


Figure 4 MS5611 thread structure

Referring to Figure 4, the green nodes are the application functions and the grey nodes are primarily operating system interface functions, low level drivers or HW interface functions. For this effort we are primarily concerned with testing the green application functions. The designator call-outs associated with the arcs encode the execution order, the conditional execution, and exceptions. All edges between nodes are unidirectional and represent function calls within specific functions. For example, an edge pointing from node function_A to node function_B represents function_A calling function_B within the same program. The edges have a label of either “D” or “X#”. “D” represents a “Direct Call”, which means that the function being pointed to is called by the function pointing to it without any conditional statements (i.e. the call happens unconditionally). The “X#” represents some kind of conditional statement or statements (i.e. “for” loop, “while” loop, “if” statement, etc.) within the software application code that may either prevent a function from being called or allow a function to be called. - Brackets { }, indicate the numeric order of each edge between nodes. The numeric order begins at 1 (one). The importance of this design information is that encodes the specified sequence behavior of the thread.

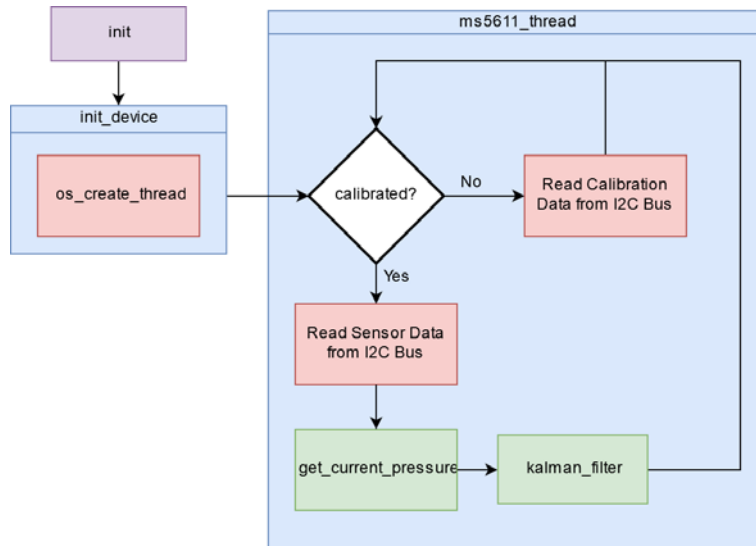


Figure 5 shows the basic flow of the ms5611_thread functions. Items in red indicate low level functions which have not been tested. Green items indicate functions that are the target for testing

The basic flow of the application thread is shown in Figure 5. The **get_current_pressure** function takes the raw temperature and pressure values from the sensor in the form of 32-bit unsigned integer values as inputs. The function outputs a double value for the current pressure value, in Pascals. The pressure calculations are performed as follows:

```

double get_current_pressure(Ms5611 *device, uint32_t currentRawPressure,
uint32_t currentRawTemperature) {
#define pressureCoefficient 0.953
    uint32_t D1 = currentRawPressure;

    uint32_t D2 = currentRawTemperature;

    int64_t tRef = (int64_t) device->reference_temperature << 8;
    int64_t dT = D2 - tRef;
    int32_t TEMP = 2000 + ((dT * device->tempSens ) >> 23);

    int64_t OFF = ((int64_t) device->pressure_offset << 16) +
        (((int64_t) (device->temp_coefficient_of_pressure_offset *
pressureCoefficient) * dT) >> 7);

    int64_t SENS = ((int64_t) device->pressure_sensitivity << 15) +
        (((int64_t) (device->temp_coefficient_of_sensitivity *
pressureCoefficient) * dT) >> 8);

    int64_t OFF2 = 0;
    int64_t SENS2 = 0;

    if (TEMP < 2000) {
        OFF2 = 5 * ((TEMP - 2000) * (TEMP - 2000)) / 2;
        SENS2 = 5 * ((TEMP - 2000) * (TEMP - 2000)) / 4;
    }

    if (TEMP < -1500) {
        OFF2 = OFF2 + 7 * ((TEMP + 1500) * (TEMP + 1500));
    }
  
```

```

        SENS2 = SENS2 + 11 * ((TEMP + 1500) * (TEMP + 1500)) / 2;
    }

    OFF = OFF - OFF2;
    SENS = SENS - SENS2;

    double pressure = (((D1 * SENS) >> 21) - OFF) / 32768.0;
    device->temperature = TEMP;
    return pressure;
}

```

The `ms5611_kalman_filter` function performs the Kalman Filter calculations in order to reduce noise. A floating-point value for the measured pressure values, prior to the Kalman Filter calculations, is used as the input, while the output is a floating-point value for the pressure values, following the Kalman Filter calculations. The Kalman Filter equations are as follows:

$$\begin{aligned}
 kf.P &= kf.P + kf.varP; \\
 kf.K &= kf.P / (kf.P + kf.varM); \\
 kf.Kalman &= kf.K * _p + (1 - kf.K) * kf.Kalman; \\
 kf.P &= (1 - kf.K) * kf.P;
 \end{aligned}$$

The individual functions in the code are as follows:

```

static float ms5611_kalman_filter(float \_p){

    kf.P = kf.P + kf.varP;
    kf.K = kf.P / (kf.P + kf.varM);
    kf.Kalman = kf.K * \_p + (1 - kf.K) * kf.Kalman;
    kf.P = (1 - kf.K) * kf.P;

    return kf.Kalman;
}

```

For both functions `get_current_pressure` and `ms5611_kalman_filter`, a test oracle was created to verify that they were functioning as expected.

4.3 TESSY and ACTS tools

There are two tools which figure promptly in our testing methodology. First, the NIST's ACTS tool performs automatic test case generation for t-way test vectors. Secondly, Razorcat's TESSY tool provides the environment to automate the execution of the tests on embedded software on the actual target. TESSY's purpose is to provide "scaffolding" and "tools" to facilitate automated testing. Two to six-way combinatorial test cases are generated using ACTS using function input variables and test values from the embedded code. The combinatorial test cases that are generated by ACTS are imported into TESSY and used to execute the unit tests for the functions.

TESSY has an inbuilt test oracle procedure for comparing the expected output values against the actual output values and evaluating the test results. Detailed test reports can be generated by TESSY that describe the test environment settings, coverage results, complexity metrics, test statistics, input domain classification tree, and test module properties. In addition, the test report also includes the input values, prolog and epilog, expected output values, actual output values, and the test results for each test step. TESSY also generates detailed coverage metrics reports for individual coverage metrics and a graphical view of coverage results. As shown in Figure 6, in TESSY, one may also set a breakpoint when the Test object is entered during test execution; TESSY also allows the use of debugger features like line steps, watch windows, etc. to evaluate variable values. The many aforementioned features of TESSY aid in easily finding the reasons behind deviation in output values from their expected values that caused Test-case failures [16].

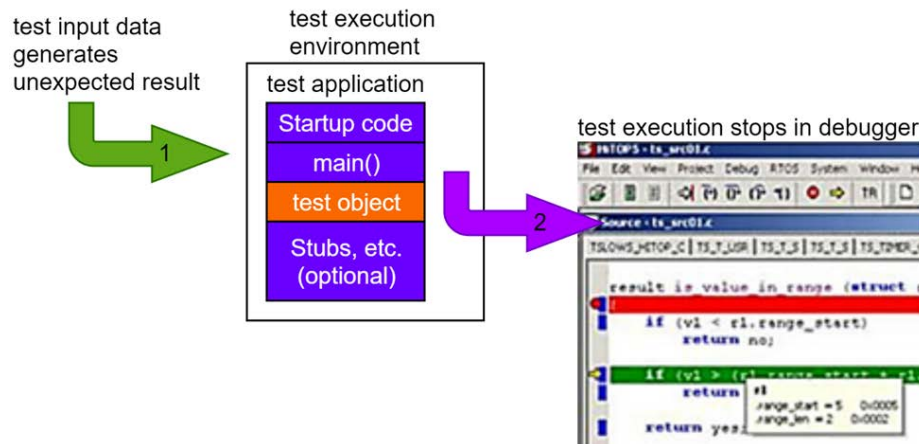


Figure 6 Breakpoint Feature - A failed test case can be debugged by using the breakpoint feature of TESSY [15]

4.4 Test Interface

As we discussed earlier, SW testing on embedded digital devices is often difficult due to the limited observability of code execution behavior and temporal nature of real time systems. Minor changes in the code due to test instrumentation (such as user instrumented variable writes), can cause task timing to deviate. In extreme cases, the test instrumentation can cause the “false” failures to occur. One of the supporting features of the VCU Smart sensor (see appendix b) is that it has On Chip Debug (OCD) capability. OCD is like a window into the execution of the SW on the CPU. Through OCD ports developers, designers, and testers are able to halt, read, write, and modify data and instructions at near CPU clock speeds – without impacting the behavior of the executing software. This allows unobtrusive “white box” testing on embedded devices. Almost all microprocessors within the last 10-15 years have some level of OCD support. Razorcat TESSY utilizes OCD software to assist in execute unit /component level tests directly on the microcontroller. As depicted in Figure 7, TESSY uses third-party OCD software (specific for the microcontroller) as interface to the test execution environment. TESSY controls this OCD software during the tests to allow remote automated control of all test case execution. The disadvantage to using OCD test interfaces for SW testing is they are somewhat complex and require some special skills to understand.

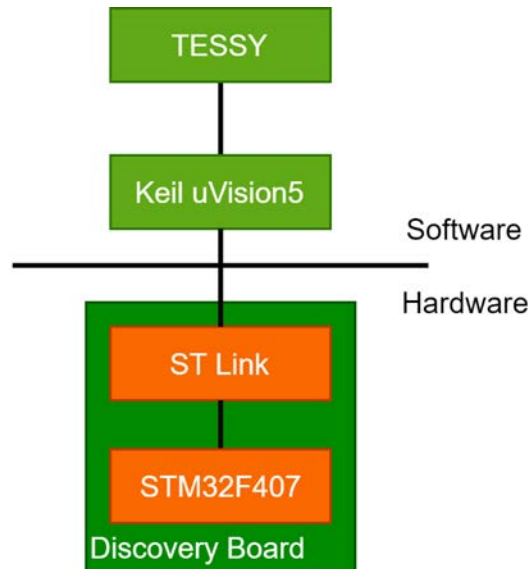


Figure 7 On Chip Debugger Interface to Smart Sensor [15]

4.5 Processes Critical to Bounded Exhaustive Testing

Our approach to bounded exhaustive testing connects and depends on several key methods. First, is well developed input models that reflect both nominal and out of range values for variables. At present this accomplished using classification tree methods to gain a better understanding of the variables and there use in the code. Secondly, is path analysis or path coverage analysis for the input test cases. A set of test vectors that does not achieve high path coverage indicates that not all paths of the software have been reached. To address this, we try to achieve maximal path coverage for all test vector sets so that confidence in the reachability of t-way tests is high. Third, is sequence and integration level testing. Sequences in function calls (at the right time and right order) within a thread must validated with robust test sets to determine if race conditions or misordering of events can occur. In addition, state-based sequences in the application have to be tested as well. We discuss these processes in more detail below.

4.5.1 Input models and Classification Tree Method

As example, Figure 9 shows the classification tree created for the input variables in `ms5611_get_current_pressure` function. There are 8 input parameters/variables used in this function that retrieves the current pressure value in Pascals. These 8 input parameters are classified based on their input ranges specified in the `ms5611` barometer device datasheet. Min, Max and Typical Values for the input variables are derived from the datasheet, given in Figure 8. In order to test for outside normal range values, variables that have invalid values within its datatype range are also classified into an outside normal range class. In this example, valid pressure and temperature data ranges between 0 and 16777215, but the datatype is unsigned int 32 which ranges from 0 to 4294967296. Hence all values greater than 16777215 are considered as invalid values. To consider this value for testing, a class 'max+1' is created which is assigned a value 16777216.

Read calibration data (factory calibrated) from PROM						
Variable	Description Equation	Recommended variable type	Size ^[1] [bit]	Value		Example / Typical
				min	max	
C1	Pressure sensitivity SENS _{T1}	unsigned int 16	16	0	65535	40127
C2	Pressure offset OFF _{T1}	unsigned int 16	16	0	65535	36924
C3	Temperature coefficient of pressure sensitivity TCS	unsigned int 16	16	0	65535	23317
C4	Temperature coefficient of pressure offset TCO	unsigned int 16	16	0	65535	23282
C5	Reference temperature T _{REF}	unsigned int 16	16	0	65535	33464
C6	Temperature coefficient of the temperature TEMPSENS	unsigned int 16	16	0	65535	28312

Read digital pressure and temperature data						
D1	Digital pressure value	unsigned int 32	24	0	16777216	9085466
D2	Digital temperature value	unsigned int 32	24	0	16777216	8569150

Figure 8 Variable value ranges as defined in the ms5611 device datasheet [17]

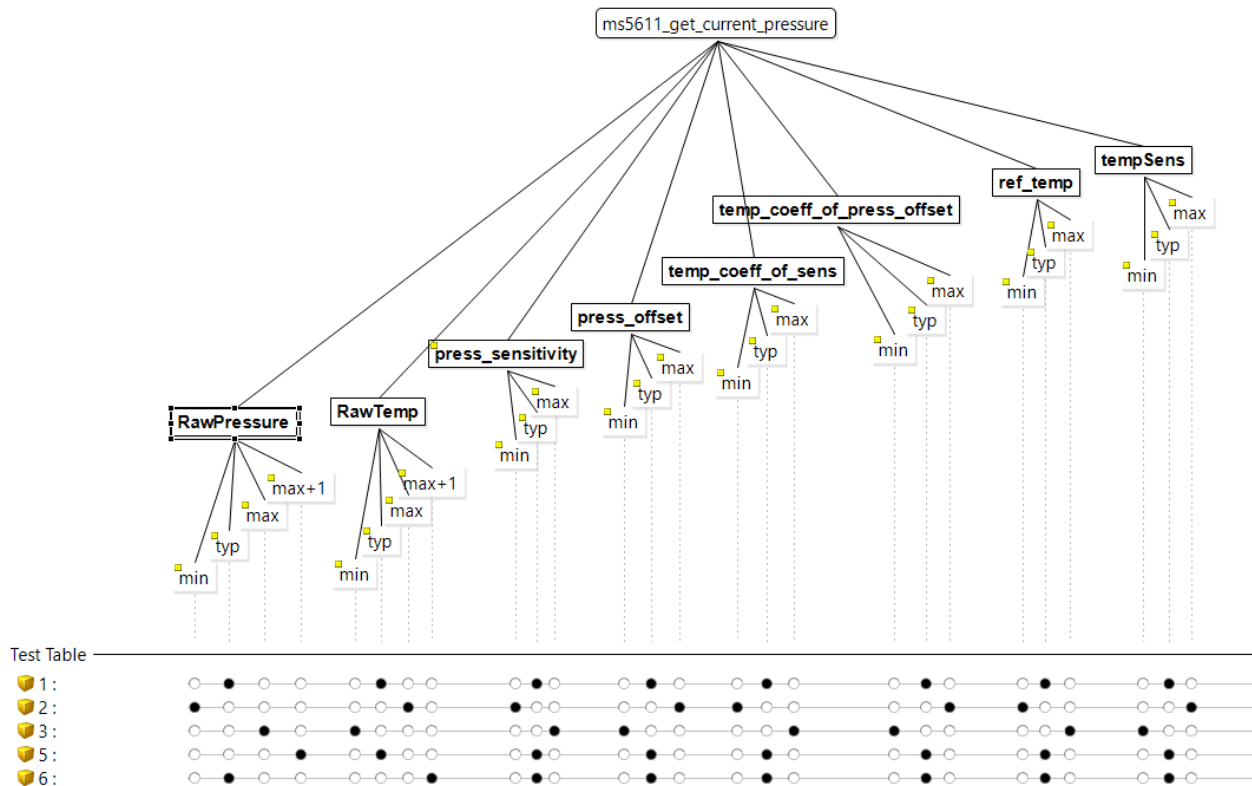


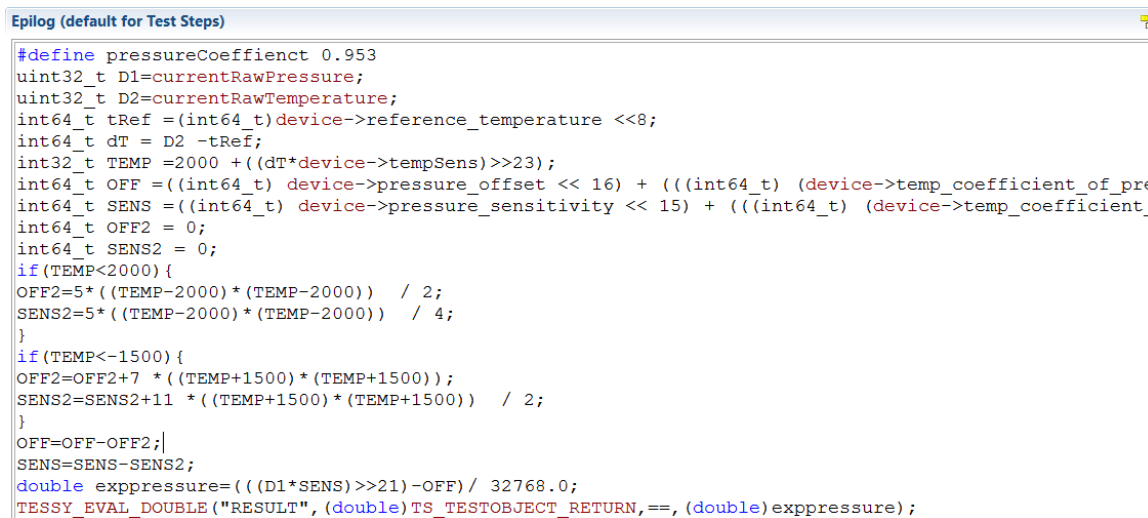
Figure 9 Classification tree for ms5611_get_current_pressure function

4.5.2 Test Oracle – from an automation perspective

A test oracle is a mechanism that determines whether software is executed correctly for a test case. Test oracle can be defined to contain two essential parts: oracle information that represents expected output; and an oracle procedure that compares the oracle information with the actual output [3]. In “Automated Software Test Implementation Guide for Managers and Practitioners, STAT COE-Report-05-2018” -

appendix E defines a number of approaches for developing oracles for SW testing. In our work, we considered a number of different approaches for oracles. From automation perspective oracles can be integrated into TESSY. In TESSY, we can provide the **oracle information** by manually writing in the expected values for the output variables for each test-case. The **Test Oracle procedure** inbuilt in TESSY compares the provided expected output values in the Test Data Editor, against the actual outputs while executing the test cases and result in Passed or Failed test-cases. However, calculating the expected output values manually is time consuming – and is not considered farther.

As shown in Figure 10, TESSY provides Prologue and Epilogue fields to fill in C code that can be executed before and after each testcase. This feature can be used to automate the calculation of the test oracle information. The algorithm for calculating the output variable values based on the input variables is derived from the requirements and written into the epilogue field for each test steps. Finally, the value calculated by the algorithm is compared with the function return value using the TESSY_EVAL_DOUBLE macro that TESSY provides. This epilogue is run for each testcase after the unit tested function is executed for each testcase.



```

Epilog (default for Test Steps)
#define pressureCoefficient 0.953
uint32_t D1=currentRawPressure;
uint32_t D2=currentRawTemperature;
int64_t tRef =(int64_t)device->reference_temperature <<8;
int64_t dT = D2 -tRef;
int32_t TEMP =2000 +((dT*device->tempSens)>>23);
int64_t OFF =((int64_t) device->pressure_offset << 16) + (((int64_t) (device->temp_coefficient_of_pre
int64_t SENS =((int64_t) device->pressure_sensitivity << 15) + (((int64_t) (device->temp_coefficient_
int64_t OFF2 = 0;
int64_t SENS2 = 0;
if (TEMP<2000) {
OFF2=5*((TEMP-2000)*(TEMP-2000)) / 2;
SENS2=5*((TEMP-2000)*(TEMP-2000)) / 4;
}
if (TEMP<-1500) {
OFF2=OFF2+7 *((TEMP+1500)*(TEMP+1500));
SENS2=SENS2+11 *((TEMP+1500)*(TEMP+1500)) / 2;
}
OFF=OFF-OFF2;|
SENS=SENS-SENS2;
double exppressure=((D1*SENS)>>21)-OFF)/ 32768.0;
TESSY_EVAL_DOUBLE("RESULT", (double)TS_TESTOBJECT_RETURN,==, (double)exppressure);

```

Figure 10 Epilogue code under each test case for ms5611_get_current_pressure function

4.5.3 Thread Level Integration Testing

Recall our approach starts with unit testing, and then incrementally builds upon thread level testing and the system level testing. To facilitate these higher-level interactions in the software, we need to develop *sequence scenarios* – temporal interactions between functions. Accordingly, thread level integration testing is performed by verifying the interactions between functions within a thread using component testing tools in TESSY. *Figure 4* shows the call graph for the ms5611_thread. It shows that the ms5611_thread function mainly has multiple external function calls that happen unconditionally or conditionally in a specific order, as given in *Table 2*. Thread level interaction testing is necessary to show that the temporal order of the functions is being preserved as intended. The results section of the document provides more details on how thread level Integration testing was accomplished.

Table 2 Invoking conditions for function calls in the call graph

Edge Label	Definition	Condition
D	Direct Call	N/A
X1	#if ARIES_MS5611_USE_MS5611D1	Edge is completed
X2	if(!device)	Returns false Edge is not completed
X3	if(ms5611_run_enabled)	Edge is completed
X4	if(!device→calibrated)	Edge is completed
X5	if(device→calibrated)	Edge is completed
X6	if(ms5611_thread_idx++>= ms5611_data_length-2)	Edge is completed

4.5.4 Test Case Coverage

Once a t-way test vector or case is created we would like to know if that test vector or case sufficiently exercises the control flow structure of the software. This helps evaluate “productive or good test cases”. A test object is considered to consist of items like branches, conditions, etc. Code coverage measures, how many of the items were exercised during the tests. This number is related to the total number of items and is usually expressed in percent.

TESSY supports seven different coverage measurements. Coverage results are visualized in a graphical flow chart linked with colorized source code views as well as in textual form. Powerful navigation through the flow chart easily reveals uncovered branches and conditions being spotlighted within the codeview [18], [19].

1. C0 (Statement Coverage) - It is a metric, which is used to calculate and measure the number of **statements** in the source code which have been executed.
2. C1 (Branch Coverage) - Branch coverage measures the coverage of both conditional and unconditional branches.
3. DC (Decision Coverage) - As depicted in Figure 11, decision coverage measures the coverage of conditional branches.

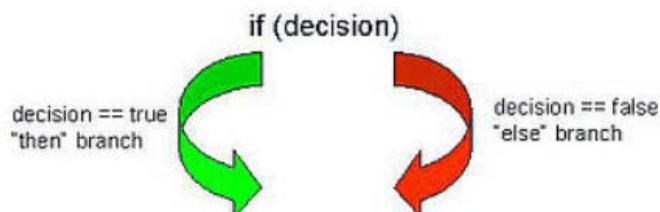


Figure 11 Decision Coverage [19]

4. MC/DC (Modified Condition / Decision Coverage) – As shown in *Figure 12*, the MC/DC coverage criterion requires each condition constituting the decision to independently change the decision outcome during the test execution.

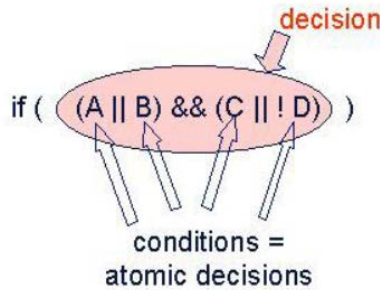


Figure 12 MC/DC Coverage [19]

5. MCC (Multiple Condition Coverage) - In **Multiple Condition Coverage** for each decision, all the combinations of true and false for the conditions should be evaluated.
6. EPC (Entry Point Coverage) - only for unit tests
7. FC (Function Coverage) - only for component tests – Checks if all functions are covered during test execution.

At present, we have selected a high level of test coverage as prescribed by IEC 61508 – SIL 3 or 4, as shown in *Figure 13*. This standard is assigned all coverage metrics Statement, Branch Decision, MC/DC, MCC and Entry point coverage.

Coverage Selection					
IEC 61508:2010 - SIL 4					Preferences...
<input checked="" type="checkbox"/>	Coverage	Name	Test Type	Minimum	^
<input checked="" type="checkbox"/>	C0	Statement Coverage	Any	100%	
<input checked="" type="checkbox"/>	C1	Branch Coverage	Any	100%	
<input checked="" type="checkbox"/>	DC	Decision Coverage	Any	100%	
<input checked="" type="checkbox"/>	MC/DC	Modified Condition / Decision C...	Any	100%	
<input checked="" type="checkbox"/>	MCC	Multiple Condition Coverage	Any	100%	
<input checked="" type="checkbox"/>	EPC	Entry Point Coverage	Unit	100%	v

Figure 13 Test coverage criteria selection

The combinatorial tests run on the function resulted in meeting 100% coverage for all metrics, as shown in

Figure 14. The coverage viewer gives graphical view of the function control flow graph by highlighting the covered paths in green and uncovered paths in red. *Figure 15* shows that both the paths at the two decision points in the function `ms5611_get_current_pressure` are taken when the testcases are run, thus being highlighted in green.

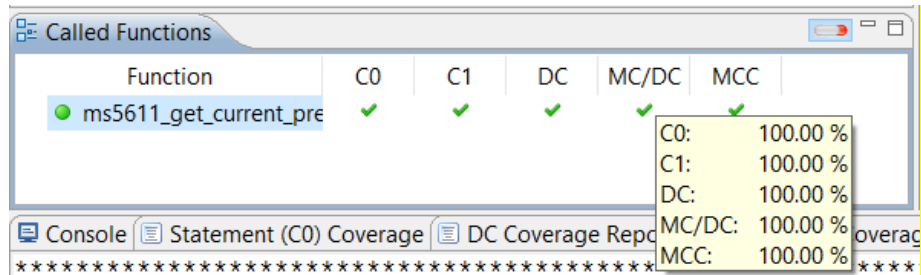


Figure 14 Test coverage with ms5611_get_current pressure combinatorial testing

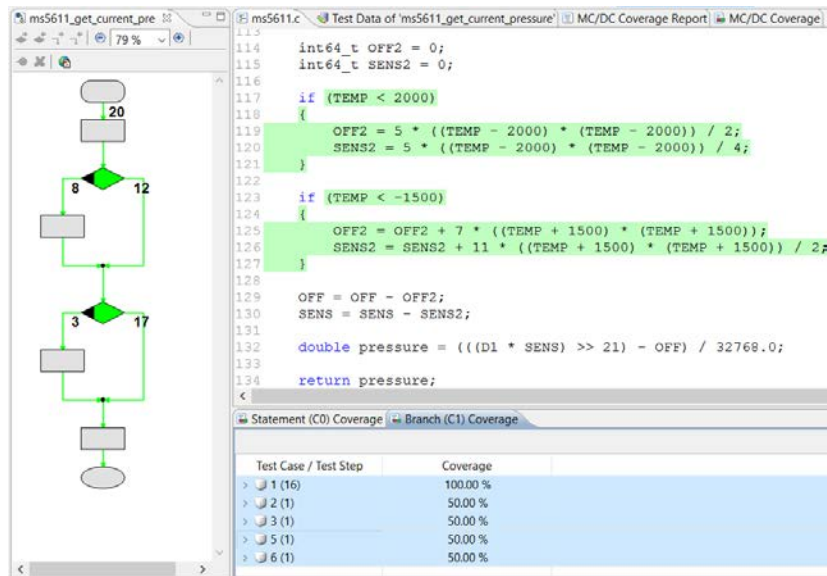


Figure 15 Test coverage shown in control flow graph of ms5611_get_current_pressure function

4.5.5 ACTS2TESSY Translator

Using the tests generated by ACTS within TESSY involves generating an ACTS input profile from the results of the boundary value analysis done in TESSY, generating the tests, and then converting them to a format that TESSY can import and subsequently execute. To accomplish this task, a Python program was written which takes an XML file with the test values produced by TESSY as well as an XLSX file which stores the tests TESSY will run, and automatically produces XLSX files with 2 to 6-way tests. Since the tests generated by ACTS are input into the same XLSX file which TESSY can import/export for testing purposes, the XLSX files with the ACTS tests can successfully be imported into TESSY.

The first step of this process is generating the ACTS input profile. When boundary value analysis is done in TESSY, TESSY will produce a test set with all combinations of inputs for the values chosen. A detailed report with these tests and values can be exported in XML format. A Python program loads this XML file and searches for all tags with test cases in them and builds a list of each variable and value found in the tests, as well as their types. Once each test has been checked, the list contains all variables and all values which are needed for generating tests. Then, the Python program builds a text file which has lines listing each variable, its type, and the possible values it can take. Once the input profile for ACTS has been generated, ACTS is run 5 times to produce t-way tests for t equaling 2 to 6, producing

CSV files containing each of the test sets. Table 3 shows how the test sets increase in size from t=2, to t=6.

The next step in the process is taking the test sets generated by ACTS and formatting them so that they can be imported into TESSY. To do this, a Python program first opens the CSV file with the tests and reads in each test case, noting the variable name that each value belongs to. The XLSX file from TESSY is opened and the existing tests in it are stripped out. Next, each of the tests is written to the XLSX file, making sure to match the column names to each of the variables. This modified XLSX file is saved as a separate file, and this process is repeated for each of the test sets generated by ACTS. When this process is complete, there are XLSX and CSV files containing all of the t-way tests needed for testing.

Table 3 Number of t-way Tests generated from ACTS tool for the functions in ms5611_thread.

Tested function	# Variables	# Values/ Variable	2-Way Tests	3-Way Tests	4-Way Tests	5-Way Tests	6-Way Tests
get_current_pressure	9	4-5	20	76	285	870	2411
kalman_filter	5	6-7	48	316	1608	7776	N/A

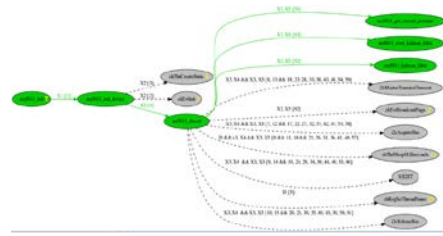
5. Preliminary Results

5.1 Overview of the Systematic Testing Approach

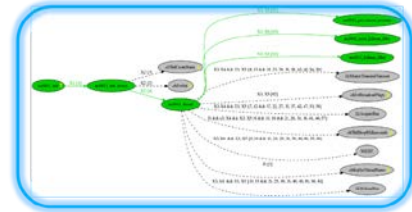
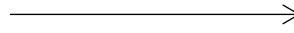
From the broadest stance, our testing approach as described in section 4 is closest to white box testing methods. T-way combinatorial testing is traditionally classified as an opaque box method – it does not require detailed understanding of the code to perform it – it only requires a detailed understanding of the inputs and variables that influence inputs. However, we analyze the code structure in our approach to maximize the potentials of BVA, t-way test sets, and path coverage. We feel white box methods are more likely to achieve compliance for regulatory guidelines (e.g. NRC BTP 7-19 or IEC 61508 Section 3).

Our approach systematic approach is based on the well-known “divide and conquer” approach of SW testing. Referring to Figure 16, our testing begins with unit testing on the functions related to each of the threads. Here we are testing interactions at the inter-functional level. After unit testing, the next level of testing is integration testing, that is, how the functions in a thread are interacting with each other. Lastly, we perform system testing where interactions across all threads occur. This systematic process ensures that the overall functional and temporal behavior of the software architecture is tested and verified. This type approach is typically required for safety critical SW with respect to standards such as IEC 61508, DO-178c, and ISO 62626.

Unit testing: on functions that are leaves in the call graphs.



Integration testing: at thread level, to test interaction between the functions within a thread.



System level testing: to test the interaction/data sharing between the threads

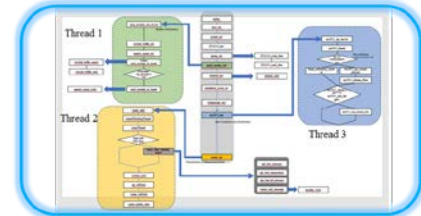


Figure 16 Testing strategy

Overlaying the “levels” of testing approach described above are the experimental study factors specifically related to this study. Referring to table 1, for some thread function “foo” we start with a baseline test level (e.g. $t=2$). We then incrementally increase levels of t to 3, 4, 5, and 6. For this preliminary study we kept input model at a baseline level, MC/DC coverage requirement is goal directed toward high (over 95%). From this basic experiment factor table, one can design studies to collect preliminary data to examine relevant questions.

- What levels of t are sufficient for detecting interaction faults?
- How does path analysis (MC/DC) coverage impact fault detection?
- How much state space is being tested or how much coverage is acquired through various levels of t ?

To collect data on all of these questions, a statistical multifactor experiment would be required to determine all possible interactions in the experiment parameters, which is beyond the scope of this work. Our goal for this phase of the work is to execute basic studies to collect data for determining the significance and power of t -way combinatorial testing for Embedded Digital Devices.

Table 4 Combinatorial Testing Experimental Study

Function	Foo 1 (initial test level)	Foo 1 (increased test level 1)	Foo 1(increased test level 2)	Foo 1(increased test level 3)	Foo 1(increased test level 4)
BVA cardinality and span – the input model (m)	Initial value for m. (for example 6 values/per foo function variables)	Enhance m by x	Enhance m by x2	Enhance m by x3	Enhance m by x4
T-way interaction level	Initial value t =2	Increment by 1	Increment by 1	Increment by 1	Increment by 1
Number of test cases	Initial value determined by initial BVA and t-way parameters	Increases as a factor of both t-way level and BVA	Increases as a factor of both t-way level and BVA	Increases as a factor of both t-way level and BVA	Increases as a factor of both t-way level and BVA
MC/DC coverage requirement	high	high	high	high	high

The development and validation of the testbed was completed in late July 2019 and beta testing started in August. As expected, a number of integration issues arose when we started beta testing. These included unit testing of SW components with infinite loops with real time RTOS, limitations on SW test cases due to memory limitations, etc. These “integration lessons and findings” will be detailed in special report. Once the test bed was validated, we initiated testing on three devices concurrently. Beta results were cross validated between devices to ensure that all three devices were behaving as expected.

5.1.1 Summary of Results

As shown in table 5 summary of results for unit testing, about ~9700 test vectors were applied to two functions; (1) **get_current_pressure**, and (2) **Kalman_filter**. As we expected, the number of test vectors increases exponentially as t increases. The variables values from BVA analysis varies from variable to variable, but the average is around 7 values/variable (see classification tree in section 4.5.1). The time to execute tests was reasonable for real time testing on a physical device – about 1 test was executed, checked and saved every 400ms. This corresponds to an “at best” test productivity about 210,000 test vectors/day. While this is significantly lower than what can be achieved in simulation, it is considered to be very good for actual testing on a physical HW device.

Table 5 Summary of Results

Function	Testing Type	Testing Method	Coverage	# Tests	Time	Pass/Fail	Number of Defects
get_current_pressure							
T=2	Unit	BVA, T-way, MC/DC	Statement=100% MC/DC=100%	21	Less 1 min	21/0	0
T=3	Unit	BVA, T-way, MC/DC	Statement=100% MC/DC=100%	76	1 min	76/0	0
T=4	Unit	BVA, T-way, MC/DC	Statement=100% MC/DC=100%	285	2.3 min	285/0	0
T=5	Unit	BVA, T-way, MC/DC	Statement=100% MC/DC=100%	870	8 min	870/0	0
T=6	Unit	BVA, T-way, MC/DC	Statement=100% MC/DC=100%	2411	15 min	2411/0	0
Kalman_filter							
T=2	Unit	BVA, T-way, MC/DC	Statement=100% MC/DC=100%	48	Less 1 min	42/6	2
T=3	Unit	BVA, T-way, MC/DC	Statement=100% MC/DC=100%	316	3 min	276/40	2
T=4	Unit	BVA, T-way, MC/DC	Statement=100% MC/DC=100%	1608	12.4 min	1389/219	2
T=5	Unit	BVA, T-way, MC/DC	Statement=100% MC/DC=100%	7776	59.1 mins	5855/1921	2
				9748			

The code tested was native code that had been used in previous applications of a UAV autopilot for years. Most notably, with this limited testing we found three defects in the two functions we tested. One of the defects we uncovered was a kind of a “buffer overflow” defect that was not expected to be found with this type of interaction testing - possibly meaning that this technique is more robust than anticipated. Some initial thought has been given to what would be the trigger to this defect, and at this time the most likely trigger would be a HW fault that triggers a additional “byte” of data to forwarded to the function. Future research would be useful to better understand how this testing method finds these types of errors, triggers and other errors beyond interaction, timing, and sequence errors.

We are careful not to overreach on these conclusions based on the findings to date. There are other SCCF fault types that would need to be tested by other methods – such as memory leaks, volatile reassignments, and certain race conditions. Dynamic SW testing methods, such as described in this report are often paired with static code testing/assessment methods to look for dead code, vulnerabilities and weaknesses in code prior deployment. It is usually good practice to perform static analysis on code prior to dynamic testing to identify and fix these issues. We are investigating using the Mathworks Polyspace static analysis tool as a companion to these testing methods. There are several qualifications to our findings which we discuss below.

- The first qualification is related to oracles and model checking for bounded exhaustive testing. In the original paper by Kuhn [7], model checking was employed to develop both test cases and oracles for the test cases. Due to time limitations, we were not able to complete this aspect of the testing strategy. We believe the approach outlined in [7] can be employed in the test strategy; however, we need additional time to investigate how to incorporate it. Also, we need to investigate how to use model checking where variables are real-valued (e.g. float or double).
- The second qualification is related to the BVA and classification tree methods for the input model. At present we have developed a “baseline” input model. More research or effort should be

applied on how we can develop input test cases that are “stressing”. One approach to stressing SW is based on fault injection practices and techniques, which both VCU and NIST have significant experience with.

- The third qualification we must mention is related to metrics for measuring the sufficiency of testing. Coverage metrics and state space metrics exist for both combinatorial and path-based testing methods. Some rudimentary metrics were presented in section 3.1.1 that can be the basis of developing metrics. Since our testing method combines aspects of combinatorial and path-based testing strategies it seems reasonable to adopt or develop a metric that provides indicators on the sufficiency of testing with respect to these methods. We intend to address metrics fully in the future addendum to this report.
- The fourth qualification is that we did not have time to test the approach against a set of reference faulty version of the code – that is seeded faults in the original code to ascertain the effectiveness of the approach to different fault classes. From previous research and internal testing, we have developed ~42 faulted versions of the code. Executing the full suite of tests on all 42 faulty versions of the code would be a significant undertaking for the remainder of the contract time (3 months). We will select a subset of these faults and perform testing with these faulted versions.

Although our testing to date is partial, the methodology and the testing strategy suggest that productive outcomes can be accomplished with bounded exhaustive t-way testing for embedded digital devices in safety critical nuclear power applications. With respect to the addressing the objective questions:

Question 1: Can bounded exhaustive t-way testing provide evidence that is congruent with exhaustive testing for an embedded digital device?

Under what assumptions and conditions for this claim to be true?

Question 2: Is t-way combinatorial testing effective at discovering logical and execution-based flaws in nuclear power software-based devices (device under test [DUT])?

Evidence to support question 1 is still preliminary as research and testing continues, but the evidence and experiences to date afford confidence that there is strong potential to answer the question in the affirmative. Specifically, the aggregate elements that comprise bounded exhaustive testing; (1) t-way combinatorial testing, (2) sequence testing, (3) path analysis, and (4) boundary value analysis provide significant coverage of the execution behavior of the software. Since it is well known that exhaustively testing the entire state space of even a simple SW system is impossibility, the approach we and others in the scientific community have encouraged is testing a program’s execution state space or behaviors. Bounded exhaustive testing is focused on testing program execution behaviors. Our goal is to complete testing in the near future to collect more findings to relate to this question.

Answering question 2 is more affirmative. We discovered three native defects in the code that were latent for sometime, and in one case the defect was there for years. Additionally, the overflow defect we uncovered was not expected to be found with this type of interaction testing – which is good indicator that the methods may have more detection potential than we expected. These preliminary outcomes suggest that bounded exhaustive testing of the type we define in this report, supported by the automation methods can be effective for addressing SCCF needed for qualification.

Some questions raised in the preliminary research:

- *What about the real time OS (RTOS) or the executive SW on the embedded digital device?* We did not test functions related to the RTOS on the VCU smart sensor, but in a real qualification exercise some evidence would be required for the integrity of the RTOS. It's our experience that developers of real time systems often partner with companies whose sole product is real time software that is certified to some integrity level (e.g. SIL). This allows the developer to focus on their application integrity and not the RTOS integrity.
- *How does one choose oracle methods or design oracles for real time testing?* We certainly struggled with this aspect of the testing, but we found plenty of material on suggested accepted approaches (e.g. "Automated Software Test Implementation Guide for Managers and Practitioners, STAT COE-Report-05-2018" - appendix E). An important aspect not to lose sight of is the real time nature of the embedded digital devices needs to be factored into the oracle design – time is important. Our way of dealing with temporal aspects of the oracle is that TESSY has timestamping built into the framework.
- *How do you trust test automation and how do you deal with the potential large volumes of test sets and data that result?* Testing of real time embedded digital devices will require well designed and coordinated automation. Know your automation. It's better to partner with a company or developer who specializes in test automation for real time systems. Acquiring testing frameworks that have a legacy of being used across multiple platforms and application domains provides confidence that tools will be able to satisfy the needs expected during testing. The downside to test automation is that can obscure the details of the test execution, and sometimes this can lead to problems later on. We had a few occasions where we assumed something about the test automation process, and we were wrong about the assumptions. In our case, it did not lead to invalidating results or hindering progress. The problem we encountered required us to perform our tests in segments rather than one continuous test set – because of the memory limitations of the embedded digital devices. Also, large amounts of data from testing are the norm. Have skilled team members or specialists in data analytics and can use data analytic programming tools like iPython, R, or Matlab.

6. Details of the Preliminary Results

This section provides a more detailed perspective of the testing and results. The section describes our approach and results for both unit testing and integration testing on the two threads tested. Explanation of the failures found are given as well.

6.1 Issues Identified in Software by Testing

Table 6 provides a summary of the defects detected by the testing approach. Three defects were identified in the preliminary testing. The first two of these defects can be ascribed to deficiency of robust software engineering practice in these functions. The last is example of the software coder or engineer interpreting the specification too literally. The engineer saw that the spec required only three bytes for data range and that the sensor head would only be sending three bytes via INT structure (4 bytes). The last byte is

ignored as it is empty. The issue is the `ms5611_get_current_pressure` function will accept INT (beyond three bytes). Again, these are native bugs and not seeded bugs.

Table 6 Software issues identified by testing

Issues	Software Function	Issue caught by
Missing Divide by Zero check	<code>ms5611_kalman_filter</code>	2-way combinatorial unit testing
Missing Overflow check in float computations	<code>ms5611_kalman_filter</code>	2-way combinatorial unit testing
Function processes input values outside valid range	<code>ms5611_get_current_pressure</code>	2-way combinatorial unit testing

6.2 Unit Testing and Results

We begin with presenting the approach and result of the unit testing on the two functions we tested, `ms511_kalman_filter` and `ms511_get_current_pressure`.

6.2.1 Function `ms5611_kalman_filter`: 2-way combinatorial testing

The below Figure 17 shows the interfaces (variables, parameters, return values etc.) that are of relevance while performing the unit testing of `ms5611_kalman_filter` function. This function `ms5611_kalman_filter` performs Kalman filtering of pressure values read from the `ms5611` barometer device. ‘`_p`’, the argument that is passed into the function is the current pressure value in Pascals that is calculated from the raw data sent out by the `MS5611` sensor head device. It is of floating-point datatype. For filtering this pressure value Kalman filter equations are used and there are five different filter parameters (`P`, `varP`, `varM`, `K` and `Kalman`) that are used in the Kalman filtering equations. These parameters are of float datatype and are members of the `kalmanFilter_t` structure.

Filter interface elements	
External Functions	Passing
Local Functions	
External Variables	
Global Variables	
float Result	IRRELEVANT
kalmanFilter_t kf	IN
float P	IN
float varP	IN
float varM	IN
float K	IRRELEVANT
float Kalman	IN
Parameter	
float _p	IN
Return	
float	OUT

Figure 17 Test Interfaces for ms5611_kalman_filter function

During unit testing these Kalman filter parameters (except K) and the floating-point pressure value become the test inputs to the function and are therefore set as 'IN's in the Test interface Editor for this function. The parameter K is calculated internally based on the other filter parameters. Hence this variable is not an input to the function and therefore marked as IRRELEVANT to testing. The return value from this function, which is the filtered pressure value, is the output to be monitored and therefore set as 'OUT'. So, there are totally 5 inputs and 1 output for testing this function. For each of these float inputs the BVA analysis and equivalence partitioning is performed using the classification tree in TESSY.

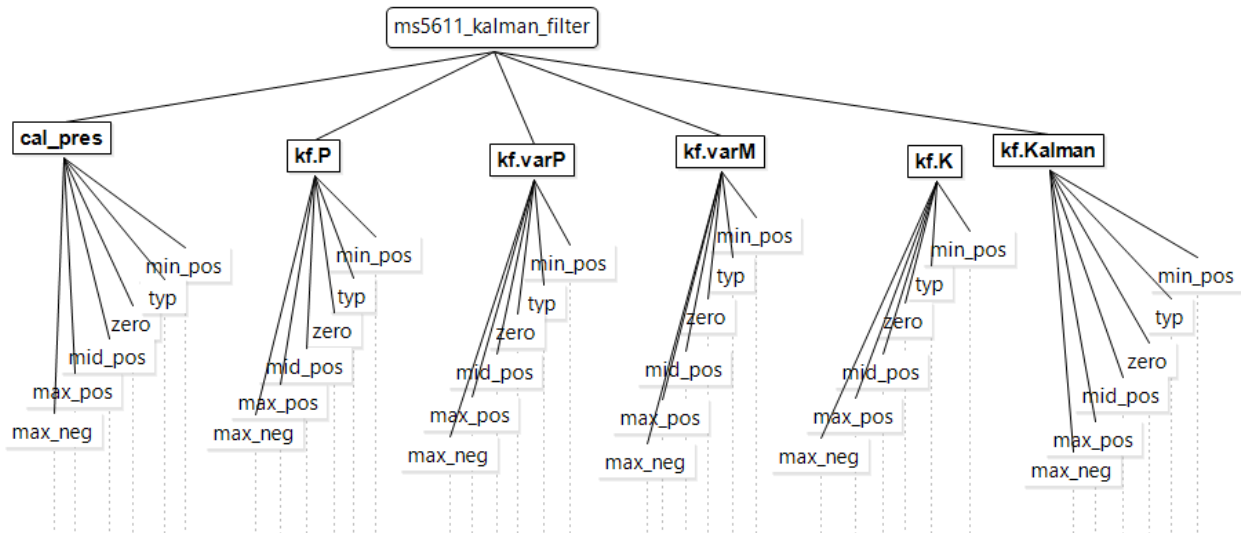


Figure 18 Classification tree for ms5611_kalman_filter function

Six classes of sample test values are obtained for each of the float inputs after BVA as shown in Figure 18. Table 7 shows the sample input values corresponding to each class in the classification tree.

Table 7 Sample input values assigned to each class in the Classification tree for ms5611_kalman_filter function

Class name	Value
Max_neg	-3.40282e+038
Max_pos	3.40282e+038
Zero	0
Min_pos	1.1755e-038
Mid_pos	65535
Typ _p (typical pressure value)	3.8
Typ P (typical value for parameter P)	1
Typ varP (typical value for parameter varP)	0.0001
Typ varM (typical value for parameter varM)	0.25
Typ Kalman (typical value for parameter Kalman)	20

Combinatorial tests generated from ACTs contain 2- to 5-way combinations of all these test values. In this case the combinatorial tests only go up to 5-way as there are only 5 input variables under consideration. The 2-way test results are presented in Table 8 below. The first testcase involves a test input combination, where the pressure value _p and Kalman filter parameter P are negative extreme values of float datatype -3.40E+38, and the other 3 inputs are 0s. Based on the Kalman filter equations given below, the expected Kalman filtered pressure value for Testcase1 also computes to -3.40E+38.

$$\begin{aligned} \text{kf.P} &= \text{kf.P} + \text{kf.varP}; & (1) \\ \text{kf.K} &= \text{kf.P} / (\text{kf.P} + \text{kf.varM}); & (2) \\ \text{kf.Kalman} &= \text{kf.K} * _p + (1 - \text{kf.K}) * \text{kf.Kalman}; & (3) \end{aligned}$$

The expected output value is calculated within the prologue code for each testcase and TESSY compares the calculated expected value with the actual return value from the function. As the expected value and actual value matches both being -3.40E+38, the testcase1 passes. Most combination of input values end up in a passed test result. However, there are 6 failed testcases. Analyzing them in detail identifies 2 potential issues in the design of the Kalman filter function which describe in the next sections.

Table 8 2-way Combinatorial test results for ms5611_kalman_filter function

Test Case	2-way Tests							
Test Step	_p	kf.P	kf.varP	kf.varM	kf.Kalman	Expected Value	Actual Value	Pass/Fail
1	-3.40E+38	-3.40E+38	0	0	0	-3.4E+38	-3.4E+38	ok
2	-3.40E+38	3.40E+38	3.40E+38	-3.40E+38	-3.40E+38	1.#QNAN0	1.#QNAN0	notok
3	-3.40E+38	0	0.0001	0.25	20	-1.4E+35	-1.4E+35	ok
4	-3.40E+38	1	1.18E-38	1.18E-38	1.18E-38	-3.4E+38	-3.4E+38	ok
5	-3.40E+38	1.18E-38	65535	65535	65535	-1.7E+38	-1.7E+38	ok
6	-3.40E+38	65535	-3.40E+38	3.40E+38	3.40E+38	1.#INF00	1.#INF00	notok
7	3.40E+38	3.40E+38	0	0.25	1.18E-38	3.4E+38	3.4E+38	ok
8	3.40E+38	-3.40E+38	0.0001	1.18E-38	65535	3.4E+38	3.4E+38	ok
9	3.40E+38	0	1.18E-38	65535	3.40E+38	3.4E+38	3.4E+38	ok
10	3.40E+38	1	65535	3.40E+38	-3.40E+38	-3.4E+38	-3.4E+38	ok
11	3.40E+38	1.18E-38	-3.40E+38	-3.40E+38	0	0	0	ok
12	3.40E+38	65535	3.40E+38	0	20	3.4E+38	3.4E+38	ok
13	0	3.40E+38	0.0001	65535	0	0	0	ok
14	0	-3.40E+38	1.18E-38	3.40E+38	20	1.#QNAN0	1.#QNAN0	notok
15	0	0	65535	-3.40E+38	1.18E-38	0	0	ok
16	0	1	-3.40E+38	0	65535	0	0	ok
17	0	1.18E-38	3.40E+38	0.25	3.40E+38	0	0	ok
18	0	65535	0	1.18E-38	-3.40E+38	0	0	ok
19	3.8	3.40E+38	1.18E-38	0	65535	3.8	3.8	ok
20	3.8	-3.40E+38	65535	0.25	3.40E+38	3.8	3.8	ok
21	3.8	0	-3.40E+38	1.18E-38	-3.40E+38	3.8	3.8	ok
22	3.8	1	3.40E+38	65535	0	3.8	3.8	ok
23	3.8	1.18E-38	0	3.40E+38	20	20	20	ok
24	3.8	65535	0.0001	-3.40E+38	1.18E-38	0	0	ok
25	1.18E-38	3.40E+38	65535	1.18E-38	20	0	0	ok
26	1.18E-38	-3.40E+38	-3.40E+38	65535	1.18E-38	1.#QNAN0	1.#QNAN0	notok

27	1.18E-38	0	3.40E+38	3.40E+38	65535	65535	65535	ok
28	1.18E-38	1	0	-3.40E+38	3.40E+38	3.4E+38	3.4E+38	ok
29	1.18E-38	1.18E-38	0.0001	0	-3.40E+38	0	0	ok
30	1.18E-38	65535	1.18E-38	0.25	0	0	0	ok
31	65535	3.40E+38	-3.40E+38	3.40E+38	20	20	20	ok
32	65535	-3.40E+38	3.40E+38	-3.40E+38	1.18E-38	0	0	ok
33	65535	0	0	0	65535	1.#QNAN0	1.#QNAN0	notok
34	65535	1	0.0001	0.25	3.40E+38	6.81E+37	6.81E+37	ok
35	65535	1.18E-38	1.18E-38	1.18E-38	-3.40E+38	-1.1E+38	-1.1E+38	ok
36	65535	65535	65535	65535	0	43690	43690	ok
37	-3.40E+38	1.18E-38	0.0001	3.40E+38	1.18E-38	0	0	ok
38	1.18E-38	65535	1.18E-38	-3.40E+38	65535	65535	65535	ok
39	3.40E+38	3.40E+38	65535	0	3.40E+38	3.4E+38	3.4E+38	ok
40	65535	-3.40E+38	-3.40E+38	0.25	-3.40E+38	1.#QNAN0	1.#QNAN0	notok
41	65535	0	3.40E+38	1.18E-38	0	65535	65535	ok
42	65535	1	0	65535	20	20.99968	20.99968	ok
43	65535	1	0.0001	1.18E-38	3.40E+38	65535	65535	ok
44	0	65535	65535	65535	-3.40E+38	-1.1E+38	-1.1E+38	ok
45	65535	1.18E-38	65535	3.40E+38	0	0	0	ok
46	0	1.18E-38	0.0001	-3.40E+38	20	20	20	ok
47	65535	0	3.40E+38	0	1.18E-38	65535	65535	ok
48	-3.40E+38	0	65535	0.25	65535	-3.4E+38	-3.4E+38	ok

6.2.2 Failures with Infinity test result

Infinity test results are obtained when the Kalman filter calculations end up in division by zero.

For example, the 6th testcase in Table 8 causes the function to return an infinity result.

With the above being the Kalman filter equations, the 6th testcase with values, $kf.P = 65535$, $kf.varP = -3.40282E+38$ and $kf.varM = 3.40282E+38$, causes the first equation $kf.P = kf.P + kf.varP$ to evaluate to $-3.40282E+38$. And the second equation $kf.K = kf.P / (kf.P + kf.varM)$ to evaluate to: $-3.40282E+38 / (-3.40282E+38 + 3.40282E+38) = -3.40282E+38 / 0 = \text{Infinity}$.

An infinity result is not a preferred result and this test failure therefore indicates that the design of the `kalman_filter` function does not have an adequate robustness check for divide by zero prevention.

6.2.3 Failures with ‘NaN’ (Not a Number) test result

‘NaN’ (Not a Number) test results are obtained when the kalman filter calculations end up in either 0/0 or Inf/Inf computations whose results are undefined. As most engineers will agree, NaN’s are the bane of SW functionality – they are difficult to detect and rarely occur during normal operations. For example, the 2nd testcase in Table 8 causes the function to return a ‘NaN’ (Not a number) result.

The 2nd testcase with values, $kf.P = 3.40282E+38$, $kf.varP = 3.40282E+38$ and $kf.varM = -3.40282E+38$, causes the first equation $kf.P = kf.P + kf.varP$ to evaluate to Infinity. And the second equation $kf.K = kf.P / (kf.P + kf.varM)$ to evaluate to:

$Inf/(Inf + -3.40282E+38) = Inf/Inf = NaN$ (or undefined).

An undefined result is not a preferred result and this test failure therefore indicates that the design of the kalman_filter function does not have a robust overflow prevention check. Having an overflow prevention check could avoid ending up in Infinity results during float computations. The consequences of these defects are difficult to determine without system context. Generally, these types of errors usually result in significant disruption of the functionality of the device.

6.2.4 Coverage Statistics

As shown in Table 9, for the kalman_filter function, the maximum path based coverage of 100% is achieved for all the coverage metrics, statement, branch, decision, MC/DC and MCC at 2-way combinatorial testing itself. Achieving 100% test coverage is not always possible, but in this case, we were able to achieve it.

Table 9 Coverage Statistics for 2-way Combinatorial testing on ms5611_kalman_filter function

kalman_filter_2-way.xml	Coverage Statistics				
name	success	total	reached	notreached	percentage
Statement Coverage (C0)	ok	2	2	0	100
Branch Coverage (C1)	ok	1	1	0	100
Decision Coverage (DC)	ok	1	1	0	100
Modified Condition / Decision Coverage (MC/DC)	ok	1	1	0	100
Multiple Condition Coverage (MCC)	ok	1	1	0	100

6.2.5 Function ms5611_kalman_filter 3-5 way combinatorial testing

As we go on to 3-way, 4-way and 5-way combinatorial testing, the test set grows in size with more input combinations. The number of test failures also increases as more and more input combinations result in Infinity and Nan results due to divide by 0 and 0/0 or Inf/Inf computations. But essentially, the root cause of the all test failures in the 3-way,4-way and 5-way testing are the same as already identified with the 2-way testing. Hence it can be stated the test failures obtained in 3-, 4- and 5-way tests are redundant. The

two issues identified in the function, missing divide by zero check and missing overflow check, are already caught in the 2-way combinatorial testing.

6.2.6 Function ms5611_get_current_pressure 2-6way combinatorial testing

The input space for this function has partitions to handle invalid values for the inputs. All the testcases within the 2-way, 3-way, 4-way, 5-way and 6-way test sets pass for this function. As given in *Figure 9* and explained in section 5.5.1, the raw pressure and raw temperature values have a class ‘max+1’ that is assigned a value (16777216) outside of its valid value range (0-16777215). Testcases with this ‘max+1’ values of raw pressure and raw temperature were expected to fail. But they ended up with Passed results. This reveals another possible issue within the design of the ms5611_get_current_pressure function. The function processes input values that are outside of their valid range whereas it was not expected to do so (e.g. this is the classic buffer overflow problem). This might result in erroneous pressure values to be sent out of the smart sensor. Coverage statistics show that for ms5611_get_current_pressure function, the maximum coverage of 100% is achieved for all the coverage metrics, statement, branch, decision, MC/DC and MCC at 2-way combinatorial testing itself.

6.3 Component/Integration Testing and Results: ms5611_thread

Sequence based tests are formulated to validate the correctness of the ms5611_thread functionality – that is validating the call graph execution sequence for all valid data flows. To execute these tests we had to develop a “tick function” – to emulate the behavior of the real time operating system. A dummy tick function is set as the work task with a call rate of 10ms – the cycle rate of the operating system scheduler. Time steps at 10ms rate are created. In each 10ms time steps, necessary local function calls are added to set variables to various values and simulate various test scenarios. External functions that are expected to be called every 10ms are added in the correct order in the Sequence Editor under each time step.

The component tests only give access to global variables and parameters passed to the local function calls. In the first execution of the ms5611_thread the calibration parameter values from the ms5611 device are read into the software. Once the calibration is done, from the next thread function execution, the pressure value is read from the device and the filtered pressure value is calculated in Pascals in the software.

The precondition for activating ms5611_thread is to set the enable parameter for the thread. Figure 20 shows that in the 10ms time step, the ms5611_set_run_enabled is called first. Thus ms5611_thread is enabled to be executed. Further the call to ms5611_thread function is made by passing the target_ptr pointer. This pointer is created in TESSY. The receive buffer that contains the i2c read raw pressure value and calibration parameters are set to specific test values.

At 0ms, when the ms5611_thread function is called, the calibrate function is called from within the ms5611_thread. The i2c functions i2c_AcquireBus, i2c_MasterTransmitTimeout, i2cReleaseBus and chThdSleep functions are expected to be called sequentially and repeatedly 7 times which includes sending the ms5611_Reset command and then reading in the 6 calibration parameters from the device.

The calibration parameters are:

- pressure_sensitivity,
- pressure_offset,
- temp_coefficient_of_sensitivity,
- temp_coefficient_of_pressure_offset,
- reference_temperature,
- tempSens.

The snapshot of the test results in Figure 19 shows that the actual function calls made at 0 ms matches both in count and order the expected function calls.


TEST DETAILS REPORT		2019-08-30, 08:37:50-0400			
smartsensorproj 2					
Actual Function	Count	Expected Function	Count	Result	
i2cAcquireBus [0 ms]	1	i2cAcquireBus [0 ms]	1	✓	
i2cMasterTransmitTimeout [0 ms]	1	i2cMasterTransmitTimeout [0 ms]	1	✓	
i2cReleaseBus [0 ms]	1	i2cReleaseBus [0 ms]	1	✓	
chThdSleep [0 ms]	1	chThdSleep [0 ms]	1	✓	
i2cAcquireBus [0 ms]	1	i2cAcquireBus [0 ms]	1	✓	
i2cMasterTransmitTimeout [0 ms]	1	i2cMasterTransmitTimeout [0 ms]	1	✓	
i2cReleaseBus [0 ms]	1	i2cReleaseBus [0 ms]	1	✓	
chThdSleep [0 ms]	1	chThdSleep [0 ms]	1	✓	
i2cAcquireBus [0 ms]	1	i2cAcquireBus [0 ms]	1	✓	
i2cMasterTransmitTimeout [0 ms]	1	i2cMasterTransmitTimeout [0 ms]	1	✓	
i2cReleaseBus [0 ms]	1	i2cReleaseBus [0 ms]	1	✓	
chThdSleep [0 ms]	1	chThdSleep [0 ms]	1	✓	
i2cAcquireBus [0 ms]	1	i2cAcquireBus [0 ms]	1	✓	
i2cMasterTransmitTimeout [0 ms]	1	i2cMasterTransmitTimeout [0 ms]	1	✓	
i2cReleaseBus [0 ms]	1	i2cReleaseBus [0 ms]	1	✓	
chThdSleep [0 ms]	1	chThdSleep [0 ms]	1	✓	
i2cAcquireBus [0 ms]	1	i2cAcquireBus [0 ms]	1	✓	
i2cMasterTransmitTimeout [0 ms]	1	i2cMasterTransmitTimeout [0 ms]	1	✓	
i2cReleaseBus [0 ms]	1	i2cReleaseBus [0 ms]	1	✓	
chThdSleep [0 ms]	1	chThdSleep [0 ms]	1	✓	
i2cAcquireBus [0 ms]	1	i2cAcquireBus [0 ms]	1	✓	
i2cMasterTransmitTimeout [0 ms]	1	i2cMasterTransmitTimeout [0 ms]	1	✓	
i2cReleaseBus [0 ms]	1	i2cReleaseBus [0 ms]	1	✓	
chThdSleep [0 ms]	1	chThdSleep [0 ms]	1	✓	
i2cAcquireBus [0 ms]	1	i2cAcquireBus [0 ms]	1	✓	
i2cReleaseBus [0 ms]	1	i2cReleaseBus [0 ms]	1	✓	
i2cMasterTransmitTimeout [0 ms]	1	i2cMasterTransmitTimeout [0 ms]	1	✓	
chThdSleep [0 ms]	1	chThdSleep [0 ms]	1	✓	
chThdSleep [10 ms]	1	chThdSleep [10 ms]	1	✓	
i2cAcquireBus [10 ms]	1	i2cAcquireBus [10 ms]	1	✓	
i2cMasterTransmitTimeout [10 ms]	1	i2cMasterTransmitTimeout [10 ms]	1	✓	
i2cReleaseBus [10 ms]	1	i2cReleaseBus [10 ms]	1	✓	
chThdSleep [10 ms]	1	chThdSleep [10 ms]	1	✓	
i2cAcquireBus [10 ms]	1	i2cAcquireBus [10 ms]	1	✓	
i2cMasterTransmitTimeout [10 ms]	1	i2cMasterTransmitTimeout [10 ms]	1	✓	
i2cReleaseBus [10 ms]	1	i2cReleaseBus [10 ms]	1	✓	
i2cAcquireBus [10 ms]	1	i2cAcquireBus [10 ms]	1	✓	

Figure 19 Snapshot of Component Test results of ms5611_thread

At 10ms, when ms5611_thread function is called, the control flow is changes. The I2C commands to convert pressure and then adc_read and convert_temperature and then adc_read are sent back to back to the ms5611 sensor device. Thus, the i2c_AcquireBus, i2c_MasterTransmitTimeout, i2cReleaseBus and chThdSleep are expected to be called sequentially and repeatedly 4 times. This control flow path is continued in the following time steps 20ms, 30ms... etc. and therefore the function call sequence is the same as in the 10ms step. As shown in Figure 20, the difference is only in the test input values in the i2c rxbuf and the calibration parameters. In the 20ms test step the raw pressure value in the i2c receive buffer is set to value exactly equal to the maximum valid pressure value of 24bits, by setting the all the 3 bytes to 255 and the 4th byte to 0. In the 10ms test step, the raw pressure value in the i2c receive buffer is set to value greater than the maximum valid pressure value of 24bits, by setting the 4th byte rxbuf[3] to 1. This test step is expected to return the same pressure value as in the 20ms step as the function is supposed to saturate the pressure value to the maximum valid value of 24bits for the pressure calculations.

All test steps during component testing on ms5611_thread passed indicating that the order and number of actual function calls made at all the time steps matched the expected function calls.

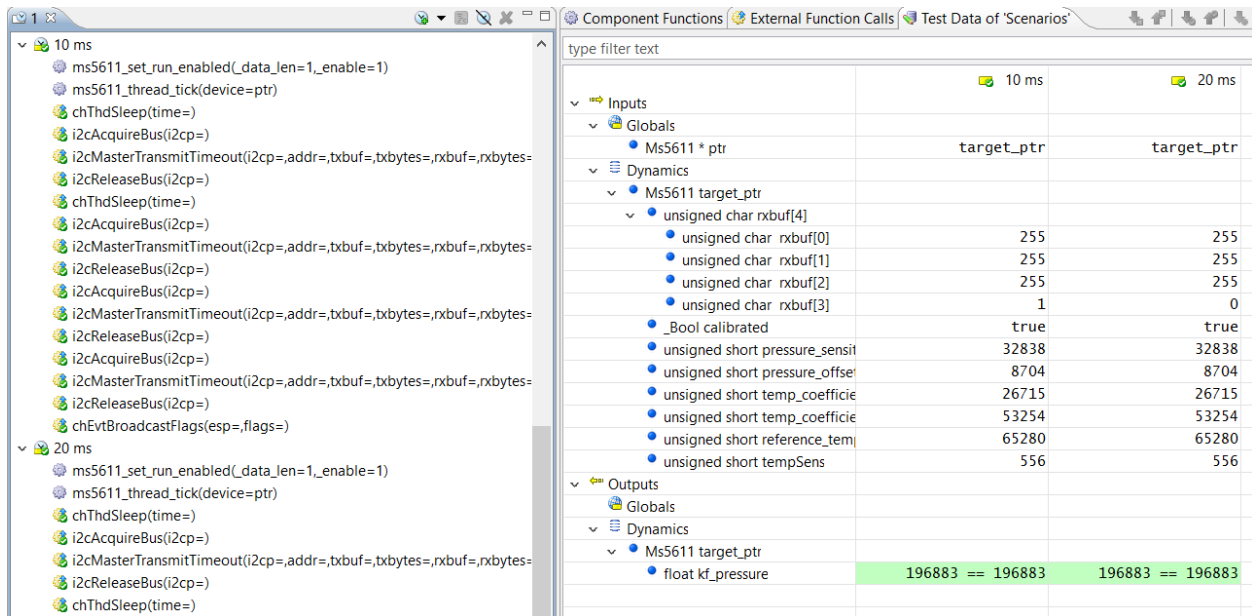


Figure 20 Sequence based testing for ms5611_thread in TESSY

7. Other Findings

The overall goal of this phase of work was to investigate bounded exhaustive t-way combinatorial testing as a suitable method for testing SW on digital embedded devices. Our guideposts for this phase of work focused on several sub-requirements.

1. Test methodology is based on established science for safety critical systems and devices.
2. All testing methods are applied directly to the embedded digital device in real time.
3. High degree of observability and controllability of execution behavior of real time SW processes and tasks.
4. Efficiency of testing is accelerated with advanced test automation methods and data collection techniques.
5. Collect preliminary results to evaluate the process.

These sub-requirements are interconnected and dependent, but we provide some insights on each of these as they were realized in the span of our research.

Test methodology is based on established science: From the outset we investigated the potential bounded exhaustive t-way combinatorial testing as way to address SCCF for embedded digital device qualification. Report 1 [20] surveyed the science and methodology behind bounded exhaustive t-way combinatorial testing and concluded the methods are well founded and supported by extensive studies and evidence. The only issue found is the small set of applications of embedded real time devices with bounded exhaustive testing. Based on this work, the preliminary findings strongly suggest that the application of bounded exhaustive t-way combinatorial testing to embedded digital devices presents no

significant challenges. This claim assumes that support for automated real time testing of the type we employed for our studies is used. The workflow and test strategy we used is highly influenced by the needs of standards such as IEC 61508 and NRC regulatory guide BTP-7-19. Both of these are established guides in the nuclear industry.

Automated SW Testing for Embedded Safety Critical Devices: A significant part of the work to date has been on automated test bed and workflow necessary to support and conduct bounded exhaustive t-way combinatorial testing. For highly critical safety devices, the safety and reliability of embedded systems can only be ensured by using certified automated testing tools – which are challenging and costly to develop for developers and vendors. We discuss a number of observations related to automation of SW testing below:

- **Tools** - In recent years, mature automated SW testing tools for safety critical systems have emerged on the marketplace (both open source and commercial) which significantly accelerate the power of SW testing. We examined a number of these early in the study which is detailed in the first report [29]. In our case, we adopted two tools to underpin our automation environment: Razorcat TESSY and NIST ACTS tools ^a. Nuclear Industry Vendors and regulators will need tools that have legacy and certification credentials for safety critical testing. TESSY is TUV certified to test systems to SIL-3/4. NIST ACTS tools have been used on numerous safety critical systems and devices.
- **Investment of time and effort** - The use of the tools requires investment in time to customize them for the specific device characteristics and data collection needs of any research or testing study. As example, the cyclic real time nature of our VCU embedded digital device required us to create a unique test strategy to capture the sequential scan time aspects of our device (see section 6.5.3). The time and effort to learn the tools should be factored into the adoption process. Some aspects of the automation process are not very automated – such as the Boundary Value Analysis which is used to create the input model of the tests. As far we can tell the efficiency and the productivity of the automation tools are more than sufficient to conduct large scale tests.
- **Simulation versus Actual testing** – Testing on the actual embedded digital device (in our case the smart sensor) is the gold standard for testing and final acceptance testing. However, testing on the actual device comes with a price – it’s usually much slower than what can be accomplished by testing the SW alone or testing with aid of simulated HW. We estimate that we can execute about 200K tests per day per device. If testing required 100’s millions of tests, we would need significant speed up. One method which we have explored and collected data on is virtual HW simulators. HW simulator like OVPsim, QEMU, and Simics can run about 100 to 1000 times faster than actual HW [21], [22]. This aspect of automation may be explored in the next phase to compare the actual results with simulated results in effort to characterize the fidelity of HW simulation versus actual physical testing.

^a <https://www.razorcat.com/en/home.html>, https://csrc.nist.gov/projects/rt_automated-combinatorial-testing-for-software

- **Oracles** - The one problem that we struggled with (that seems universal to SW testing) is selection and development of a trusted oracle. We examined a number of very promising approaches (see NIST -800-142 appendix A) but settled on just a few. Our first approach was to use a diverse version of the Kalman filter using the prolog and epilogue functions in TESSY. The prolog and epililog functions allow you to encode the acceptance conditions derived from the requirements – not the source code. At present this seems to be OK but may not be ideal for more complex systems. The next approach is based on models of what the SW is supposed to do – in that sense it is a model-based testing approach. With this method, we could develop a Simulink Model of the smart sensor application, a Kalman filter with calibration. Then we would automatically generate C code from the Simulink model using the Simulink tool C-coder and embed the oracle code into TESSY. The last approach we consider uses a model checker (nuSMV) to generate checking conditions for the tests. This approach is discussed in detail in [23].

Support for Testing Real Time Devices: The main challenge in testing real-time systems is that they need to be tested in the temporal domain as well as the value domain. Testing in the temporal domain has several implications. The input to the function or thread must be issued at a precise time instance – the beginning of the scan cycle. The temporal state of the SW object at the start of the test execution must be controlled. The timing of the result at the end of execution must be observed. As detailed in section 6.1.7 for testing real time function sequencing, these tests can be complicated to develop. It requires detailed knowledge of the control flow sequencing. Control Flow Graph knowledge may not always be available to testers at the outset, but it is possible to obtain CFG information with certain analysis tools (e.g. IdaPro - <https://www.hex-rays.com/products/ida/>) . In addition, testing real time embedded devices requires lower levels of control and observability than typical non-real time SW. They also require some notion of a time base. Based previous work with embedded On Chip Debugging (OCD) interfaces [24], [25], we recognized these interfaces could provide the type of controllability and observability required for testing real time SW. An emerging idea for accelerating testing is to separate real time sensitive tasks from non-real time logical application tasks. Logical application tasks are dependent on the RTOS to establish timing of functions and data passing. This seems like a natural partition to exploit. Specially, it may be possible and beneficial to run test cases on the application part of the code on simulated hardware that can execute the tests orders of magnitude beyond what is possible on the smart sensor. The timing sensitive tests would remain on the smart sensor.

Multi-pronged approach: Our approach does not rely on t-way testing alone; rather it combines several types of testing strategies to address various challenges. In fact, most safety critical standards like IEC 61508 suggest diverse and synergetic testing methods to apply to critical software. T-way testing supported by boundary value analysis and path coverage analysis is a powerful method for detecting interaction and sequence dependent timing faults. The synergism between these methods provides a substantial coverage of the path and statement executions, the interactions of input and state variables with respect to path executions, and sequence dependencies. In our testing approach, other test case generation approaches such as mutation testing, fault injection may be added without disruption to test strategy – we created the workflow and the environment to be modular and extensible - “plug and play”.

Collect Preliminary data: Data collect so far is providing insight on evidence produced and how it can be used to qualify an embedded digital device. We are still developing post data processing methods to support the metrics we want to use for assessment. But the typical raw data is:

- How many tests passed and failed?
- Why they failed
- The MC/DC coverage
- The function tested
- The BVA test specification
- The details of t-way tests executed
- The oracle used to determine goodness of result

As we move to larger scale tests, we will need to implement post processing to filter data to support data analytics. At present we are using iPython to post process data. We may do this through a data base query tool like MySQL in the future.

7.1 Next steps

The next steps to this phase of work are to complete the study and collect data to draw well-formed conclusions to the questions raised in the objectives and scope section of this report. This will require carrying out many more tests and experiments to collect statistically significant data to support conclusions.

Potential Next Steps (subject to project authorization)

1. Additional testing on this DUT – more functions and more threads, to broaden the test results on which to base conclusions.
 - a. Development of more robust oracles with model checkers
 - b. Stress testing with fault injection
 - c. Development or adoption of relevant metrics to indicate sufficiency of testing.
2. Possibly test a commercial device, confirming that the bounded exhaustive test methodology is effective and transferable to this class of digital devices.
3. Continue to enhance and vet the testbed, in consideration of how a commercial supplier would conduct this type of device qualification testing.
4. Conduct a peer review of these initial results with experts within and outside the nuclear industry.
5. Conduct discussions with the NRC on current regulatory guidance with respect to this digital device qualification methodology.
6. Explore potential commercialization paths.

The preliminary findings up to this point suggest that bounded exhaustive t-way testing can be an effective testing strategy for embedded systems and digital devices.

8. REFERENCES

- [1] J. Offutt and P. Ammann, *Introduction to software testing*. Cambridge University Press Cambridge, 2008.
- [2] R. W. Butler and G. B. Finelli, “The infeasibility of quantifying the reliability of life-critical real-time software,” *IEEE Trans. Softw. Eng.*, vol. 19, no. 1, pp. 3–12, 1993.
- [3] J. B. Goodenough and S. L. Gerhart, “Toward a theory of test data selection,” *IEEE Trans. Softw. Eng.*, no. 2, pp. 156–173, 1975.
- [4] U. N. R. Commission, “Guidance for Evaluation of Diversity and Defense-in-Depth in Digital Computer-Based Instrumentation and Control Systems,” *Branch Tech. Position*, pp. 7–19, 2007.
- [5] “Protecting Against Common Cause Failures in Digital I&C Systems of Nuclear Power Plants,” 28-Feb-2019. [Online]. Available: <https://www.iaea.org/publications/8151/protecting-against-common-cause-failures-in-digital-ic-systems-of-nuclear-power-plants>. [Accessed: 10-Sep-2019].
- [6] C. R. Elks, A. Tantawy, R. Hite, S. Gauthem, and A. Jayakumar, “Defining and Characterizing Methods, Tools, and Computing Resources to Support Bounded Exhaustive Testability of Software Based I&C Devices,” Idaho National Lab.(INL), Idaho Falls, ID (United States), 2018.
- [7] D. R. Kuhn and V. Okum, “Pseudo-exhaustive testing for software,” in *2006 30th Annual IEEE/NASA Software Engineering Workshop*, 2006, pp. 153–158.
- [8] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, “An evaluation of exhaustive testing for data structures,” Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, 2003.
- [9] D. Coppit, J. Yang, S. Khurshid, W. Le, and K. Sullivan, “Software assurance by bounded exhaustive testing,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 328–339, 2005.
- [10] K. J. Hayhurst, *A practical tutorial on modified condition/decision coverage*. DIANE Publishing, 2001.
- [11] R. Wieringa, *Design Methods for Reactive Systems : Yourdon, Statemate, and the UML*. Boston: Morgan Kaufmann, 2003.
- [12] M. Brčić and D. Kalpić, “Combinatorial testing in software projects,” in *2012 Proceedings of the 35th International Convention MIPRO*, 2012, pp. 1508–1513.
- [13] D. Ri. Kuhn, J. M. Higdon, J. F. Lawrence, R. N. Kacker, and Y. Lei, “Combinatorial Methods for Event Sequence Testing,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Montreal, QC, Canada, 2012, pp. 601–609.
- [14] R. C. Bryce and C. J. Colbourn, “A density-based greedy algorithm for higher strength covering arrays,” *Softw. Test. Verification Reliab.*, vol. 19, no. 1, pp. 37–53, 2009.
- [15] “TESSY - Test System - Razorcat Development GmbH.” [Online]. Available: <https://www.razorcat.com/en/product-TESSY.html>. [Accessed: 11-Sep-2019].
- [16] “TESSY User Manual.” Razorcat, Feb-2019.
- [17] “MS5611-01BA03 Barometric Pressure Sensor, with stainless steel cap,” p. 22.
- [18] “Hitex: Code Coverage in TESSY.” [Online]. Available: <https://www.hitex.com/tools-components/test-tools/TESSY/code-%20coverage-in-TESSY/>. [Accessed: 10-Sep-2019].
- [19] “TESSY: Hitex: Dynamic Module/Unit Test.” [Online]. Available: <https://www.hitex.com/tools-components/test-tools/TESSY/>. [Accessed: 02-Jun-2019].
- [20] C. Elks, C. Deloglos, D. Athira Jayakumar, A. Tantawy, R. Hite, and S. Guatham, “Specification of a Bounded Exhaustive Testing Study for a Software-based Embedded Digital Device,” 2018.
- [21] “Virtual Platforms.” [Online]. Available: http://www.ovpworld.org/about_virtualplatforms. [Accessed: 11-Sep-2019].
- [22] F. E. I. Derenthal, C. R. Elks, T. Bakker, and M. Fotouhi, “VIRTUALIZED HARDWARE ENVIRONMENTS FOR SUPPORTING DIGITAL I&C VERIFICATION,” p. 13, 2017.
- [23] D. R. Kuhn, R. Kacker, and Y. Lei, “Automated combinatorial test methods: Beyond pairwise testing,” *Crosstalk J. Def. Softw. Eng.*, vol. 21, no. 6, pp. 22–26, 2008.

- [24]B. Vermeulen and K. Goossens, *Debugging systems-on-chip: communication-centric and abstraction-based techniques*. Springer, 2014.
- [25]M. Miklo, C. R. Elks, and R. D. Williams, “Design of a high performance FPGA based fault injector for real-time safety-critical systems,” in *2011 IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2011, pp. 243–246.
- [26]J. D. Elmore, “Design of an All-In-One Embedded Flight Control System,” 2015.
- [27]G. L. Ward, *Design of a small form-factor flight control system*. Virginia Commonwealth University, 2014.

APPENDIX A

EXAMPLE TEST REPORTS

Unit Level - 2-Way Combinatorial Test Results for get_current_pressure() function

name	success	total	reached	notreached	percentage
Statement Coverage (C0)	ok	5	5	0	100
Branch Coverage (C1)	ok	4	4	0	100
Decision Coverage (DC)	ok	4	4	0	100
Modified Condition / Decision Coverage (MC/DC)	ok	4	4	0	100
Multiple Condition Coverage (MCC)	ok	4	4	0	100

Note: The variable names shown below were shortened for readability.

Test Step	Var 0	Var 1	Var 2	Var 2.1	Var 2.2	Var 2.3	Var 2.4	Var 2.5	Var 2.6	Expected Value	Actual Value	Pass/Fail
1	9085466	8569150	target	32838	8704	26715	53254	65280	556	116294	116294	ok
2	9085466	8569150	-	0	65535	0	65535	0	65535	-258666	-258666	ok
3	9085466	16777216	-	65535	0	65535	0	65535	0	283925	283925	ok
4	9085466	0	-	32838	8704	26715	53254	65280	556	107228	107228	ok
5	9085466	16777215	-	0	0	26715	65535	65535	556	-0.44428	-0.44428	ok
6	0	8569150	-	65535	8704	0	0	65280	65535	-50845.4	-50845.4	ok
7	0	16777216	-	32838	65535	65535	53254	0	0	-334074	-334074	ok
8	0	0	-	0	0	0	53254	65535	65535	140466	140466	ok
9	0	16777215	-	65535	65535	26715	0	0	556	-131070	-131070	ok
10	16777216	8569150	-	32838	0	65535	65535	65280	0	-101028	-101028	ok
11	16777216	16777216	-	0	8704	26715	0	0	65535	389936	389936	ok
12	16777216	0	-	65535	65535	0	53254	65535	0	596211	596211	ok
13	16777216	16777215	-	32838	8704	65535	65535	65535	65535	245307	245307	ok
14	16777215	8569150	-	0	65535	26715	53254	65280	0	-230243	-230243	ok
15	16777215	16777216	-	65535	0	0	65535	0	556	274464	274464	ok
16	16777215	0	-	32838	8704	65535	0	65535	556	-754236	-754236	ok
17	16777215	16777215	-	32838	8704	0	53254	65280	65535	244503	244503	ok
18	0	0	-	0	8704	65535	65535	0	0	-17408	-17408	ok
19	16777216	16777216	-	65535	0	0	0	65280	556	524280	524280	ok
20	16777215	8569150	-	32838	0	65535	65535	65535	556	-104012	-104012	ok
21	9085466	16777215	-	65535	0	0	53254	0	0	80912.5	80912.5	ok

Alias	Actual
Var 0	currentRawPressure
Var 1	currentRawTemperature
Var 2	device
Var 2.0	target_device
Var 2.1	pressure_sensitivity
Var 2.2	pressure_offset
Var 2.3	temp_coefficient_of_sensitivity
Var 2.4	temp_coefficient_of_pressure_offset
Var 2.5	reference_temperature
Var 2.6	tempSens

Unit Level - 2-Way Combinatorial Test Results for kalman_filter() function

name	success	total	reached	notreached	percentage
Statement Coverage (C0)	ok	2	2	0	100
Branch Coverage (C1)	ok	1	1	0	100
Decision Coverage (DC)	ok	1	1	0	100
Modified Condition / Decision Coverage (MC/DC)	ok	1	1	0	100
Multiple Condition Coverage (MCC)	ok	1	1	0	100

Test Step	_p	kf.P	kf.varP	kf.varM	kf.Kalman	Expected Value	Actual Value	Pass/Fail
1	-3.40E+38	-3.40E+38	0	0	0	-3.40282E+38	-3.40282E+38	ok
2	-3.40E+38	3.40E+38	3.40E+38	-3.40E+38	-3.40E+38	1.#QNAN0	1.#QNAN0	notok
3	-3.40E+38	0	0.0001	0.25	20	-1.36058E+35	-1.36058E+35	ok
4	-3.40E+38	1	1.18E-38	1.18E-38	1.18E-38	-3.40282E+38	-3.40282E+38	ok
5	-3.40E+38	1.18E-38	65535	65535	65535	-1.70141E+38	-1.70141E+38	ok
6	-3.40E+38	65535	-3.40E+38	3.40E+38	3.40E+38	1.#INF00	1.#INF00	notok
7	3.40E+38	3.40E+38	0	0.25	1.18E-38	3.40282E+38	3.40282E+38	ok
8	3.40E+38	-3.40E+38	0.0001	1.18E-38	65535	3.40282E+38	3.40282E+38	ok
9	3.40E+38	0	1.18E-38	65535	3.40E+38	3.40282E+38	3.40282E+38	ok
10	3.40E+38	1	65535	3.40E+38	-3.40E+38	-3.40282E+38	-3.40282E+38	ok
11	3.40E+38	1.18E-38	-3.40E+38	-3.40E+38	0	0	0	ok
12	3.40E+38	65535	3.40E+38	0	20	3.40282E+38	3.40282E+38	ok
13	0	3.40E+38	0.0001	65535	0	0	0	ok
14	0	-3.40E+38	1.18E-38	3.40E+38	20	1.#QNAN0	1.#QNAN0	notok
15	0	0	65535	-3.40E+38	1.18E-38	0	0	ok
16	0	1	-3.40E+38	0	65535	0	0	ok
17	0	1.18E-38	3.40E+38	0.25	3.40E+38	0	0	ok
18	0	65535	0	1.18E-38	-3.40E+38	0	0	ok
19	3.8	3.40E+38	1.18E-38	0	65535	3.8	3.8	ok
20	3.8	-3.40E+38	65535	0.25	3.40E+38	3.8	3.8	ok

21	3.8	0	-3.40E+38	1.18E-38	-3.40E+38	3.8	3.8	ok
22	3.8	1	3.40E+38	65535	0	3.8	3.8	ok
23	3.8	1.18E-38	0	3.40E+38	20	20	20	ok
24	3.8	65535	0.0001	-3.40E+38	1.18E-38	0	0	ok
25	1.18E-38	3.40E+38	65535	1.18E-38	20	0	0	ok
26	1.18E-38	-3.40E+38	-3.40E+38	65535	1.18E-38	1.#QNANO	1.#QNANO	notok
27	1.18E-38	0	3.40E+38	3.40E+38	65535	65535	65535	ok
28	1.18E-38	1	0	-3.40E+38	3.40E+38	3.40282E+38	3.40282E+38	ok
29	1.18E-38	1.18E-38	0.0001	0	-3.40E+38	0	0	ok
30	1.18E-38	65535	1.18E-38	0.25	0	0	0	ok
31	65535	3.40E+38	-3.40E+38	3.40E+38	20	20	20	ok
32	65535	-3.40E+38	3.40E+38	-3.40E+38	1.18E-38	0	0	ok
33	65535	0	0	0	65535	1.#QNANO	1.#QNANO	notok
34	65535	1	0.0001	0.25	3.40E+38	6.8051E+37	6.8051E+37	ok
35	65535	1.18E-38	1.18E-38	1.18E-38	-3.40E+38	1.#QNANO	1.#QNANO	notok
36	65535	65535	65535	65535	0	43690	43690	ok
37	-3.40E+38	1.18E-38	0.0001	3.40E+38	1.18E-38	0	0	ok
38	1.18E-38	65535	1.18E-38	-3.40E+38	65535	65535	65535	ok
39	3.40E+38	3.40E+38	65535	0	3.40E+38	3.40282E+38	3.40282E+38	ok
40	65535	-3.40E+38	-3.40E+38	0.25	-3.40E+38	1.#QNANO	1.#QNANO	notok
41	65535	0	3.40E+38	1.18E-38	0	65535	65535	ok
42	65535	1	0	65535	20	20.99968	20.99968	ok
43	65535	1	0.0001	1.18E-38	3.40E+38	65535	65535	ok
44	0	65535	65535	65535	-3.40E+38	-1.13427E+38	-1.13427E+38	ok
45	65535	1.18E-38	65535	3.40E+38	0	0	0	ok
46	0	1.18E-38	0.0001	-3.40E+38	20	20	20	ok
47	65535	0	3.40E+38	0	1.18E-38	65535	65535	ok
48	-3.40E+38	0	65535	0.25	65535	-3.40281E+38	-3.40281E+38	ok

Thread Level - Sequence Test Results

name	success	total	reached	notreached	percentage
Statement Coverage (C0)	notok	230	30	200	13.04
Branch Coverage (C1)	notok	152	20	132	13.15
Decision Coverage (DC)	notok	80	11	69	13.75
Modified Condition / Decision Coverage (MC/DC)	notok	89	11	78	12.35
Multiple Condition Coverage (MCC)	notok	89	11	78	12.35
Function Coverage (FC)	notok	84	14	70	16.66

Function Declaration	C0	C1	C2	MCC	MCDC	Mccabe
int __ARM_isfinitef(float)	1	1	1	0	1	1
int __ARM_isfinite(double)	1	1	1	0	1	1
int __ARM_isinfff(float)	1	1	1	0	1	1
int __ARM_isinf(double)	1	1	1	0	1	2
int __ARM_islessgreaterf(float,float)	1	1	1	0	1	2
int __ARM_islessgreater(double,double)	1	1	1	0	1	2
int __ARM_isnanf(float)	1	1	1	0	1	1
int __ARM_isnan(double)	2	2	2	2	2	2
int __ARM_isnormalf(float)	1	1	1	0	1	2
int __ARM_isnormal(double)	1	1	1	0	1	2
int __ARM_signbitf(float)	1	1	1	0	1	1
int __ARM_signbit(double)	1	1	1	0	1	1
double _sqrt(double)	1	1	1	0	1	1
float _sqrtf(float)	1	1	1	0	1	1
double copysign(double,double)	1	1	1	0	1	1
float copysignf(float,float)	1	1	1	0	1	1
float fabsf(float)	1	1	1	0	1	1
long double fmal(long double,long double,long double)	1	1	1	0	1	1
long lrintl(long double)	1	1	1	0	1	1
long long llrintl(long double)	1	1	1	0	1	1
long lroundl(long double)	1	1	1	0	1	1
long long llroundl(long double)	1	1	1	0	1	1
long double nanl(const char *)	1	1	1	0	1	1
long double remquo(long double,long double,int *)	1	1	1	0	1	1
void prio_insert(Thread *,ThreadsQueue *)	1	2	2	4	3	3
void queue_insert(Thread *,ThreadsQueue *)	1	1	1	0	1	1
Thread * fifo_remove(ThreadsQueue *)	1	1	1	0	1	1
Thread * lifo_remove(ThreadsQueue *)	1	1	1	0	1	1
Thread * dequeue(Thread *)	1	1	1	0	1	1

APPENDIX B – VCU Smart Sensor

This appendix provide background to the VCU Smart Sensor which is the target Embedded Digital Device under test for this study.

1. Introduction

The VCU Smart Sensor is a barometric pressure and temperature sensing device that originates from the VCU UAV Laboratory. The device is derived from a Part 23 (non-safety related) VCU ARIES_2 Advanced Autopilot Platform [26], [27], which consists of mature design and code, and has over 10,000 hours of tested flight time. The VCU Smart Sensor is comprised of both hardware and software articles, which are described more in-depth in the following sections. The definitive descriptions of the VCU Smart Sensor software are the VCU Software Requirements Specification and Software Design Document – both found on the VCU Github repository. The VCU github repository (See link below) contains all software, documentation, fault files, and testing setups. All software for the VCU Smart Sensor is written in GNU11 C programming language for the application code and compiled and executed by the GNU GCC Compiler version 7.3 to run on top of the ChibiOS Version 17.6.4 Real-Time Operating System (RTOS). The VCU Smart Sensor aims to aid in the qualification and licensing of EDDs in the Nuclear Digital I&C Domain, where the tests performed thereafter will serve as a benchmark for the originally planned CCF measurements and tests.

1.1 General Matter

The VCU Smart Sensor software consists of several threads executing periodically in a real time OS. The following generalities are mentioned to place the software development process and documentation in context.

- Development - Several developmental tools were evaluated and used in order to generate the VCU Smart Sensor and the associated supporting documentation files. The primary development tools used include the GNU11 C programming language, the GNU GCC Compiler Version 7.3.
- Code support – Our code support includes the graphic visualization tools code2flow, Doxygen, and Graphviz. The entire structure and functionality of the VCU Smart Sensor is comprised of source code, written in the GNU11 C programming language, which is readily available to the user for further inspection or external testing. The GNU GCC compiler is the standard compiler used to compile and execute the application code.
- Function Maps - The associated function-call maps, which are found in the Software Requirements Specification(SRS) and Software Design Document (SDD) reports in the VCU Gitlab repository, are generated directly from the VCU Smart Sensor application code using the online interactive code to flowchart converter, code2flow. Additionally, the open source tools Doxygen and Graphviz were used to create visual documents. Doxygen is the standard tool for generating documentation from annotated application code sources, and Graphviz is an open source graph visualization software.

1.2 User Documentation

The following documents are provided for the user for more in-depth information:

- The VCU Software Requirements Specification Document (Github)
- The VCU Software Design Document (GitHub)
- Product Specifications Document (Datasheet) for ST STM32F405xx and SM32F407xx ARM Cortex-M4, 2016.
- Reference Manual for ST STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 Advanced ARM®-Based 32-Bit MCUs, 2017.
- Product Specifications Document (Datasheet) for TE Connectivity MS4525DO PCB Mounted Digital Output Transducer, Combination Differential, Gage, Absolute, Compound, & Vacuum Temperature and Pressure Sensor with I²C or SPI Protocol, 2016.
- Product Specifications Document (Datasheet) for TE Connectivity Sensor Solutions MS4525DO PCB Mounted Digital Output Transducer, 2016.
- I2C and SPI Interface Specifications Document (Datasheet) for TE Connectivity Sensors, Interfacing to MEAS Digital Pressure Modules, 2016.
- Product Specifications Document (Datasheet) for TE Connectivity Sensor Solutions MS5611-01BA03 Barometric Pressure Sensor, with stainless steel cap, 2017.

1.3 VCU Smart Sensor Components

1.3.1 Hardware Architecture

The hardware architecture of the VCU Smart Sensor is seen in figure 1. The VCU Smart Sensor is based on the pre-existing and vigorously tested VCU ARIES_2 Advanced Autopilot Platform that has been adapted to operate on a STM32F429-Discovery board. Figure 4 shows the actual VCU Smart Sensor. The STM32F429-Discovery board uses the STM32F429ZIT6 ARM Cotrex-M4 168 MHz MCU featuring 2 MB of Flash Memory and 256 KB of RAM. Also included in the package are 64-Mbit SDRAM, an on-board ST-Link/V2 debugger, a 2.4" QVGA TFT LCD, and various dedicated communication busses.

The STM32F429-Discovery board was selected because of its low cost, versatility in use, and room for future additions to the software. Additionally, the ARM STM32FM429 System on a Chip (SoC) is a very widely used chip in the embedded systems world and in safety-related embedded systems, and there is a vast array of supporting documentation for the STM32FM429 microcontroller. The ARIES_2 was originally designed as a generic hardware and software platform, with a specific emphasis on ease of extendibility as a general computing device for embedded applications that require sensory input. More generally, ARIES_2 was designed for further platform modifications and testing.

The sensor heads for pressure and temperature is integrated using an Arduino Nano communicating through the I2C bus to simulate the functions of the MS5611-01BA Barometric Pressure Sensor. This method was chosen to allow a user to input their own pre-determined input to test and verify that the sensor drivers are operating as expected. Another benefit of this is to make the type of sensor interchangeable to apply to any future use that this platform may require.

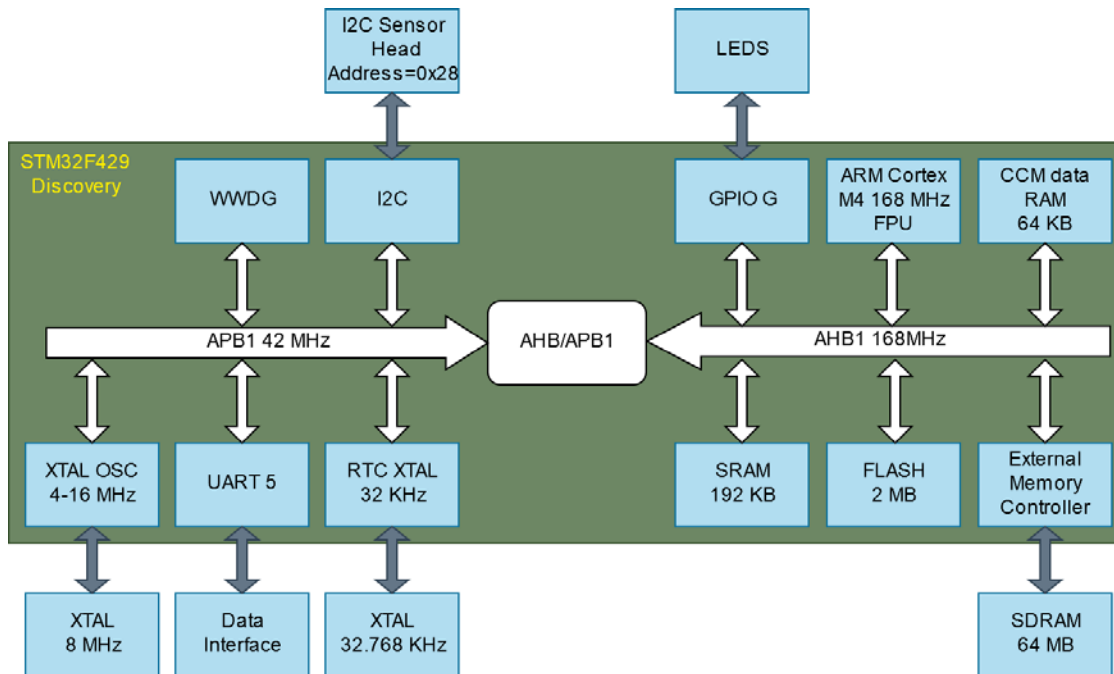


Figure1. Hardware Architecture of the VCU Smart Sensor

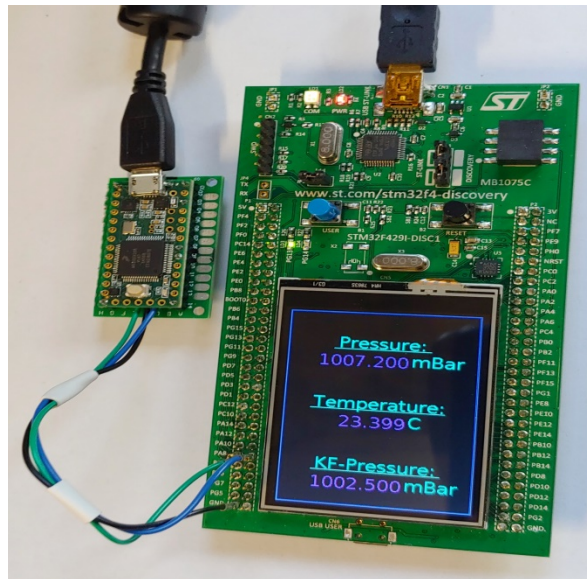


Figure2. The VCU smart Sensor Platform

1.3.2 Software Stack Model

The software stack model for the VCU Smart Sensor is seen in Figure 3. The VCU Smart Sensor software incorporates the ARIES_2 software, which has an integrated configuration system that allows for the runtime configuration of most low-level and high-level drivers, for on-board peripheral configuration.

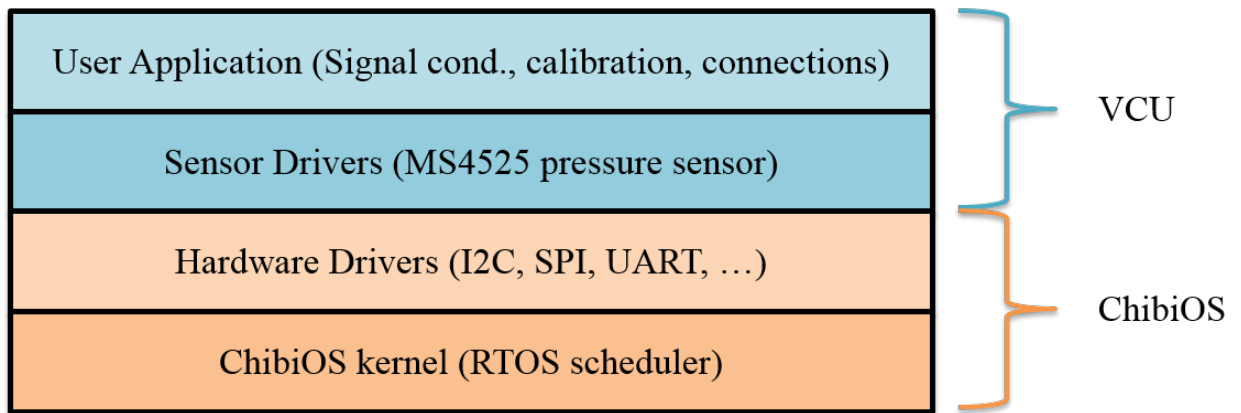


Figure 3 VCU Smart Sensor Software Stack Model

The software layers of the VCU Smart Sensor include the user application layer and the sensor drivers layer, which are original articles generated by the VCU UAV Laboratory, and modified for this current project for adjustment to the appropriate context and functionality required. The sensor drivers layer includes the drivers for the MS4525DO pressure sensor. The MS4525DO pressure and temperature transducers are managed by the user application layer in order to obtain pressure and temperature information, including altitude, speed, and offset. The VCU Smart Sensor is built around the ChibiOS real-time operating system (RTOS). ChibiOS provides the hardware drivers layer and the ChibiOS kernel layer, which include the drivers used for I2C and serial UART communication, and the RTOS scheduler, respectively.

All of the various software layers communicate with each other via I2C communication, and are managed by the full stack RTOS ChibiOS. The combination of the application code and ChibiOS ensure the proper scheduling and execution of all periodic tasks within the system, which are handled by a priority-based queue system. The software stack model has been designed and adjusted through years of implementation in order to ensure that it is a modular software design that may be adjusted or modified for future work as necessary.

1.3.3 Real-Time Operating System – ChibiOS

The software provided by VCU is built around the ChibiOS complete development environment for embedded applications. The ChibiOS development environment includes a RTOS, a hardware abstraction level (HAL), various peripheral drivers, support files and tools. ChibiOS is a free, open source RTOS, which includes many standard APIs used for most common peripherals. Additionally, ChibiOS supports the STM32FM4 and all on-board peripherals, which was the primary reason for the original design choice of using the ChibiOS development environment. Since the VCU Smart Sensor originates from the VCU

ARIES_2 Advanced Autopilot Platform, which is also built around the ChibiOS development environment, all protocols for the accurate interfacing of software using ChibiOS within the VCU Smart Sensor are already in place and have been tested extensively.

ChibiOS is a static rate based multi-threaded RTOS, which allows for deterministic behavior. The ChibiOS architecture is composed of an application model, startup code, ChibiOS/RT and ChibiOS/HAL. The application model is a single application with multiple threads, consisting of a trusted runtime environment and multiple threads that share the same address. The original RTOS scheduler has been replaced by a thread-based protocol, which generates threads during platform initialization. The generated threads are awoken as needed, either by various VCU Smart Sensor functions, or on a periodic basis using internal timers, depending on the thread. The application and operating system are linked together into a single memory image (a single program). The startup code is executed after the reset, and is responsible for core, stack, and runtime initializations, as well as the calling of the main function of the application. More information on the ChibiOS open source development environment may be found at <http://www.chibios.org/dokuwiki/doku.php>.

1.4 Smart Sensor Threads

The VCU Smart Sensor runs as a single interface application. On startup, the following will occur:

- I/O Initialization – A user-defined ASCII formatted input file for the sensor head will be fed into the Arduino from a host computer via a USB connection. This will be discussed more thoroughly in Section **Error! Reference source not found.**
- Thread Initialization – The ChibiOS Kernel will be started and the main program will become a thread. Various threads exist for specific functions within the source code.
- Serial Initialization – The Serial I/O interface will be started between the host computer and the Arduino via a USB connection. This will be discussed more thoroughly in Section **Error! Reference source not found.**
- Timer Initialization – The system timer will be initialized and started, counting the system time from system startup.

1.4.1 Data Flow

Figure 4 shows the program data flow of the software components of the VCU Smart Sensor in its testing environment context, including the threads and communication protocols used to transmit data between modules.

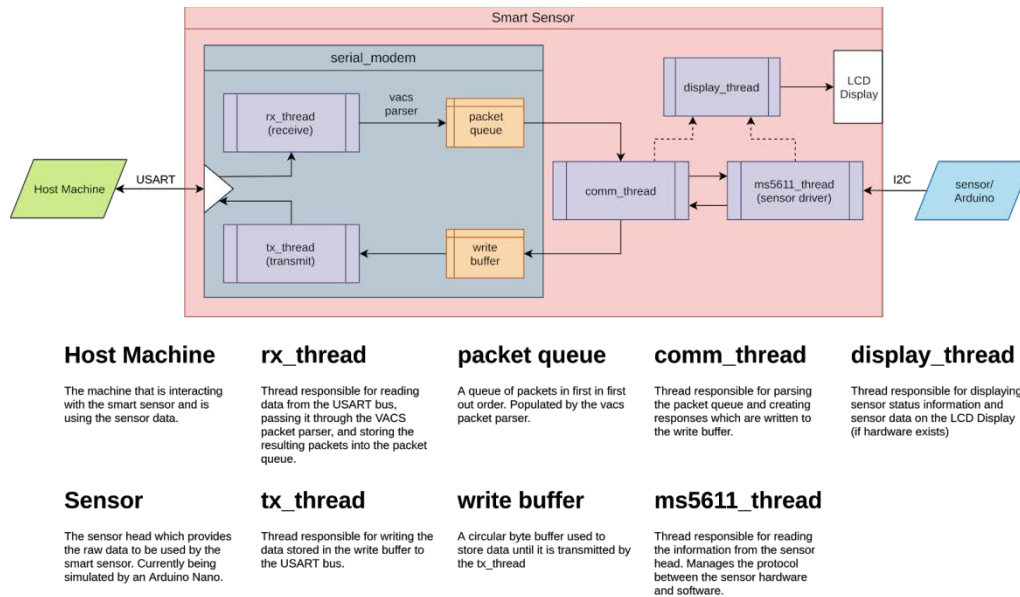


Figure 4. Program Data Flow

As seen in figure 4, the board peripherals are initialized prior to any other actions. Following the board peripheral initializations, three threads shall be generated:

- Thread #1: Serial Port
- Thread #2: Communication Transmit/Receive
- Thread #3: Barometric Sensor

Following the generation of the three respective threads, various data transmission, receive, and wait functions shall be utilized to read/write barometric data/packets (at a rate of 2Hz), calibration registers, and receive packets using serial communication protocols.

1.4.2 Communications Interfaces

All I2C communication is performed using standard I2C protocol; that is, all I2C communication is event-driven and uses write-on-requests. All communication within the VCU Smart Sensor, including the different layers of the software stack and the implementation of high-level communications functions for the API, which shall be performed using the standard I2C protocol.. The communication interface to the VCU Smart Sensor from an external user is a serial port using a serial monitor application at 57600 Baud (bits per second).

1.4.3 User Data Logging Interface

The VCU Smart Sensor shall provide a means for the logging of raw sensor data, the viewing of the data, and the downloading of the data. This interface shall be implemented via a serial port (UART) on the VCU Smart Sensor. Data will be formatted as ASCII text transmitted via RS-232 protocol to a PC-based command line prompt shell. The commands for interrogating the data are as follows:

- Initiate Data Stream
- Stop Data Stream
- Change Rate of Data Stream

1.4.4 Debug and Test Port Interface

The VCU Smart Sensor provides a debug and testing port to allow for real time monitoring of execution behavior of the VCU Smart Sensor. The VCU smart sensor will use ARM CoreSight Debug and Trace debug standard for this purpose. At a minimum, the VCU Smart Sensor will use the Serial Wire Debugger port for communicating test and debug information to commercial debug environments. A variety of debugger SW tools exist for the testing and debugging of the VCU Smart Sensor via Serial Wire Debugger. The options include the GNU GDB (GNU Debugger) (<https://www.gnu.org/software/gdb/>), the ARM Keil Microcontroller Development Kit (MDK) Toolset (<http://www2.keil.com/mdk5/>), the ARM CoreSight Debug and Trace – Serial Wire Debugger (<https://developer.arm.com/products/system-ip/coresight-debug-and-trace/coresight-architecture/serial-wire-debug>), and the Atollic Serial Wire Viewer (<http://blog.atollic.com/cortex-m-debugging-introduction-to-serial-wire-viewer-svv-event-and-data-tracing>).